A STRUCTURE PROGRAM EDITOR FOR THE
GYPSY VERIFICATION ENVIRONMENT

Dwight Francis Hare

August 1979 ICSCA-CMP-16
ACKNOWLEDGEMENTS

I would like to express my gratitude to Don Good and the entire CMP project staff for the help, encouragement, and resources which allowed me to conceive and execute this thesis. Special thanks go to Kathy Richardson for her help in preparing this thesis and to Rich Cohen and Judy Merriam for their contributions.
ABSTRACT

The Gypsy Structure Editor allows modification of Gypsy programs in the Gypsy Verification Environment. The Gypsy Verification Environment supports the verification of Gypsy programs by parsing a Gypsy program into an internal form, generating verification conditions from the specifications and code in the program, and proving the verification conditions. All of the operations required to perform these activities are supported in a uniform way inside the verification system. The Gypsy Structure Editor allows the user to modify the internal representation of the Gypsy program maintained by the Gypsy Verification Environment. This editor uses knowledge about the structure of Gypsy programs to provide powerful editing commands to the user. The editor preserves the syntactic correctness of the program at all times while helping the user to develop a syntactically correct program.
# TABLE OF CONTENTS

1.0 A STRUCTURE EDITOR IN A PROGRAMMING ENVIRONMENT . . . 8  
   1.1 Software Tools And The Programming Environment . 8  
   1.2 Program Editing In A Programming Environment . . . . . . 15  
   1.3 Overview Of The Gypsy Structure Editor . . . . . . . . . . . 18  
   1.4 Example Edit Session . . . . . . . . . . . . . . . . . . . . . . . 22  
2.0 REVIEW OF CURRENT SYSTEMS . . . . . . . . . . . . . . . . . . . . 27  
   2.1 Text Editors Versus Structure Editors . . . . . . . . . . . . . 27  
   2.2 The Interlisp Editor . . . . . . . . . . . . . . . . . . . . . . . 31  
   2.3 A General Program Editor . . . . . . . . . . . . . . . . . . . . 34  
   2.4 A Pascal Programming Environment . . . . . . . . . . . . . . 37  
   2.5 A Program Constructor . . . . . . . . . . . . . . . . . . . . . . 39  
3.0 THE GYPSY VERIFICATION ENVIRONMENT . . . . . . . . . . . . . . 41  
   3.1 The Gypsy Language And Its Internal Form . . . . . . . . . . . 41  
   3.2 Program Development, Verification, And Testing . . . . . . . 45  
4.0 THE GYPSY STRUCTURE EDITOR . . . . . . . . . . . . . . . . . . . . 47  
   4.1 The Syntax Directed Editor . . . . . . . . . . . . . . . . . . . . 47  
   4.2 The Structure Editor . . . . . . . . . . . . . . . . . . . . . . . . 53  
   4.3 The Structure Editing Functions . . . . . . . . . . . . . . . . . . 54  
   4.4 The Table Directed Editing Algorithm . . . . . . . . . . . . . . 63  
   4.5 The User Interface . . . . . . . . . . . . . . . . . . . . . . . . . . 75
5.0 EDITOR PROBLEMS AND EXTENSIONS ...............84
   5.1 The Token Editor As Text Editor ...............84
   5.2 Failures Of The Grammar Definition ............86
   5.3 The Powerful Uses Of A Structure Editor .......88
APPENDIX A - The Gypsy Grammar Definition ........96
APPENDIX B - The Editor Grammar Definition ........107
BIBLIOGRAPHY ...........................................113
CHAPTER ONE

A STRUCTURE EDITOR IN A PROGRAMMING ENVIRONMENT

A structure editor is a task specific programming tool which uses its knowledge of the editing environment to give increased power and versatility to the user. The structure editor has knowledge about the syntax and perhaps the semantics of the target programming language. This programming tool can be a vital component in a complete programming environment which is the sum of the software development tools available to apply to a programming task. A language specific structure editor gives many advantages over the alternative of a text editor much as a compiler gives many advantages over the alternative of machine language programming. Both a compiler and a structure editor are software tools which are not absolutely necessary in order to create a computer program, but whose importance for proper development and maintenance is becoming apparent.

1.1 Software Tools And The Programming Environment

Since the advent of programmable digital computers, software tools have been slowly developed to aid in the writing of programs. This began with the creation of
assemblers and relocating linking loaders until eventually high level languages were designed and compilers were implemented. The trend has been toward removing the burden of routine bookkeeping tasks from the programmer and shifting that burden to the computer. The bookkeeping tasks which have been taken over by the computer range from keeping track of symbolic addresses (assemblers) to allocating registers and translating expressions (compilers).

Until recently, however, little effort has been made to bring all of these software tools together into a complete programming environment. The advantages of an integrated system of software tools over the current scattering of isolated tools are the discovery of neglected components and the minimizing of overhead due to program modification. One of the neglected components discovered in the development of the Gypsy verification environment was the structure editor.

1.1.1 Program Development

Program development usually proceeds in a cyclic manner with the stages of design, specification, implementation, and testing. In top-down programming, for example, the top level structure is designed and, if the
programming environment allows it, it is implemented and analyzed. Then another cycle is done and the next level is designed and analyzed. If sufficient analysis or testing of the implementation can be done during this process, then later stages of the design can be influenced and improved. Viewing program development as a continuing process involving many varying stages of work reveals the cohesive nature of a complete programming environment as an aid in this process.

The Gypsy Verification Environment, which will be described more fully in chapter 3, has taken this approach toward a programming environment. This system allows incremental development, program analysis through specification and verification, testing through program interpreting, and now allows program modification with a syntax directed structure editor.

1.1.2 Incremental Development

One of the possible major benefits of a complete programming environment is intelligent incremental development. Some compilers allow compilation of individual routines which are later linked by a linking loader. Depending on the sophistication of the loader, inconsistencies undetected by the compiler might be able to
appear in the final compiled form. In these cases, it is up to the user to do the necessary bookkeeping and checking to insure against such errors. These incremental compilers are incapable of handling modifications to global constraints whose values are compiled in-line, or modifications to user type definitions.

An incremental capability, if designed into a comprehensive programming environment, can do all bookkeeping required in the modification of any program component. Using knowledge of the semantic definition of the programming language, it is possible for the system to compute the effects of modifications before they are done and thereby inform the user beforehand as to the magnitude of the work involved in order to make the modification. For example, if the user wishes to modify the parameter list of a routine, then the system knows that all routine calls to this routine must be checked for consistency with the new definition. The incremental system can inform the user prior to the change as to the number of routines which would have to be checked after the change has been made. The incremental system can enforce checking of these routines before further program analysis or testing continues - thereby ensuring against invalid results.

Much of the incremental development is not as straightforward as the example given above. Very often the
effects of modifications to a routine depend completely on which parts of the routine are modified. A modification to an internal part of a routine which is not externally visible has much less effect than a modification to an externally visible part such as a parameter list. The phrase "externally visible" means that the part is used in the checking of another routine such as checking the agreement between the parameters of a routine call and its definition.

Since the overhead involved in incremental development is dependent on the modifications made, it is important for the incremental system to know exactly which parts of a program are modified. If the changes are made to an externally maintained textual copy of the routine by a text editor and then the new version presented to the incremental system, it may be extremely difficult or impossible to discover the actual changes made.

If, however, the changes are made via another component of the programming system, a structure editor, then the editor can use its knowledge of the programming language to record the exact modifications made to the routine by the user and can allow the incremental system to use this information to minimize the overhead of incremental development. Close cooperation between the editor and the incremental system can even allow the user to be informed
about the incremental ramifications of the smallest editing change being contemplated while in the editor. Extreme ramifications of a change might indicate to the user that the proposed modification is a very basic design modification and might indicate that more thought should be given to the proposed change.

1.1.3 Program Analysis

Program analysis is greatly simplified by the existence of a comprehensive programming environment. Analysis in the form of program specification and verification can greatly enhance the correctness of a program and can isolate problems while still in the design phase.

Analysis which yields various statistics about the program can provide understanding about the program, indicate bottlenecks in efficiency considerations, and reveal violations of user enforced programming rules such as the use of abstract types in a language which does not support them. These statistics, in the form of such things as cross references, can aid in program documentation.

Such program analysis can be easily supported in an integrated way by a comprehensive programming environment.
1.1.4 System Maintenance and Documentation

Any large program development which is being implemented and maintained by a group of programmers must have a well defined set of rules governing that development in order to avoid pandemonium. A systematic programming environment can provide and enforce these rules and can provide tools to aid in the development.

These rules can include access rights for individual programmers to modify certain portions of the program. It can enforce or suggest the updating of documentation pertaining to the modified program segment and require a certain amount of analysis or testing before allowing the proposed modifications.

Developmental modifications can also be made easier by system-provided high level modifying commands. For example, changing the name of a routine involves changing the definition and all calls to that routine in the program code. A programming environment can support that kind of change as a simple command which generates little or no overhead in the system. In this sense, the modifications allowed transcend the ordinary view of a program as a sequence of characters in a text file and encourage a view of the program as a structured entity obeying rules of form and substance.
The structure editor allows many of the previously mentioned functions to be performed. The editor can restrict editing access at any level in the program. It can provide simultaneous access to those system components needed to enforce or encourage updated documentation, analysis, or testing. The editor can even maintain documentation by recording the modifications made, by whom the modifications were made, and when the modifications were made. The editor can keep a detailed enough record of the editing transactions to allow recovery of a previous version of the program. The structure editor, as part of a comprehensive programming environment, can be a powerful tool for proper system maintenance.

1.2 Program Editing In A Programming Environment

During program development, some form of the program representation must be changed to reflect desired modifications in the program. This is usually done with some form of editor.

1.2.1 Text Editors

Normally, a conception of the program as a sequence of routines, statements, and comments is transferred to a text file which becomes the unique permanent representation
of the program. Modifications are made to the program through modifications made to the text file, usually via a text editor.

This method has the advantage of using a representation which is simple to create, modify, and store. Since it is a text file, all of the software tools available to manipulate text files may be used. The most significant of these available software tools is a text editor, of which many different kinds are available.

The storage of the representation and its modification are separate from the rest of the existing programming environment. This has the advantage of being more secure from errors introduced by the programming environment since most computer systems have more reliable fail safe mechanisms than recently-devised programming environments. The form of the program kept in a text file can be structured to the desires of the programmer and will remain that way since the text file is the only representation of the program.

The most popular advantage of the use of a text file and text editor is the known mapping from the desired modifications to the sequence of editing commands on the character string. Most users of a computer system have learned how to use one or more text editors and have come to
feel very comfortable in their use. When a program modification is envisioned, the corresponding modification to the text file using the editor is usually straightforward, if not simple. Popularity is based on individual familiarity with the various text editors.

1.2.2 Structure Editors

For the purposes of analysis and other reasons, the representation of the program is translated into an internal structured form inside the programming environment. If this internal form is the permanent unique form of the program, then modification of the program requires an editor which can modify this internal form. This kind of editor is not required if a duplication of the program representation is maintained, for example, a representation in a text file. Duplication of the representation, however, allows inconsistencies to arise between the representations.

The advantages of editing this internal form using a syntax directed structure editor include efficiency and power. If the internal form is edited instead of a text form of the program, then the new internal form of the program already exists and need not be recreated from the text form. This can save tremendously in overhead for small modifications to the program.
Since the editor is aware of the syntax of the program description of the program being edited, the editor can give immediate feedback on the correctness of the syntax of the modifications being installed. It is thus possible for the editor to maintain the syntactic correctness of the program throughout the editing process.

Using knowledge of the structure of the language, the structure editor can provide complex manipulations of the program which would be very difficult using a text editor on a text file. With time, it should become as easy, or easier, to accomplish modifications using the structure editor over the text editor because the editing commands allowed in the structure editor are designed toward performing common program modifications.

1.3 Overview Of The Gypsy Structure Editor

The Gypsy Structure Editor is a syntax directed editor for the computer language Gypsy [Good 78]. It modifies the internal prefix form of the program maintained by the Gypsy Verification Environment (see Chapter 3) and provides a user interface for the display and modification of program units.
1.3.1 Structure Editor

The Gypsy program is maintained in an internal form similar to a syntax tree in some respects. The internal form was designed for the purposes of verification condition generation and proof before the Gypsy editor was ever envisioned. This means that the editor had to be designed and implemented around an existing language and internal structure form.

The structure editor must constantly resolve the two representations of the program, the internal form and the Gypsy language form which the user views. Therefore, all user interactions require a conversion from the internal form to the Gypsy language form and back again. The editor displays the pertinent internal form as a segment of Gypsy text and accepts Gypsy text from the user, translating it back into the internal form.

Since the user views Gypsy text, editing commands are represented as modifications of that Gypsy text. However, the actual modifications must be made to the internal form. Hence, the editor must convert all proposed Gypsy text modifications into the appropriate modifications of the internal form. The editor has the additional burden of maintaining the consistency of the internal form. A proposed modification to the Gypsy text might translate into
modifications to the internal form which would leave the internal form in an "illegal" state as defined by the programming environment. The internal form, as defined in the Gypsy verification environment, does not have any states corresponding to a segment of Gypsy text containing syntax errors. Therefore, this editor cannot allow any changes which would introduce syntax errors into the program since the corresponding internal form would be undefined. This constraint reduces the flexibility of the editor, but it also provides increased information and feedback to the user and prevents undesirable program conditions.

The internal representation of the program is in a more structured form than the textual representation, and the editor has knowledge about that structure. Because of this, the editor can provide complex manipulations of the program which are appropriate considering the structure of the language. The corresponding modifications to the internal form in these cases are more simple and straightforward than the corresponding modifications to the textual form.
1.3.2 Syntax Directed Editor

The Gypsy System Editor is syntax directed in the way it displays Gypsy text to the user and in the way it accepts Gypsy text from the user. The Gypsy editor presents a segment of a Gypsy program in a manner which follows the grammar definition for that segment. Each program segment corresponds to a syntax grammar production consisting of terminal and non-terminal symbols. When the segment is displayed, it is presented in a manner corresponding to the grammar production where non-terminal symbols are replaced by the appropriate program segment. The editor is directed from one part of the program to another via movements through the grammar definition.

User input is sent through the system parser which checks syntax and translates from Gypsy text to the internal form. User input is normally directed toward a particular non-terminal in the grammar production currently being displayed. The grammar definition of that non-terminal directs the parser in the translation of the Gypsy text supplied by the user.
1.4 Example Edit Session

Editing PROCEDURE COMPUTE_SUM

Edit? ?

Commands are: <escape>, Abort, DElete, DOWn, EXEc, EXIT,
Grammar, Help, Insert, Lisp, Quick, Replace,
SAve, SHow, UP, UNdo, Workspace, ^E, ?, ??.

Edit? help

For which commands would you like help? ?

Commands are: <command-list>, ALL, ^E, ?, ??.

For which commands would you like help? help

HELP

The HELP command explains the various editor commands. It takes one argument, either "ALL", which prints all of the commands for which information exists, or a specific command name, which prints the information on that command.

Edit? $

1) procedure COMPUTE_SUM (...) =
2) begin ... end

Edit? down 1

Edit? $

1) { <UNIT ACCESS LIST> } Procedure
2) COMPUTE_SUM
3) (...)
4) { <FORMAL COND PARAMS> }

Edit? show

procedure COMPUTE_SUM (A : SMALL INTEGER ARRAY;
                        I, J : ARRAY_INDEX;
                        var S : LARGE_INTEGER)

Edit? grammar

<PROCEDURE HEADER> ::= <UNIT ACCESS LIST>
                      Procedure <IDENTIFIER>
                      <PROCEDURE DATA PARAMETERS>
Edit? down 3

Edit? $
$
$
(  
1)  A : SMALL_INTEGER_ARRAY  
2)  I : ARRAY_INDEX  
3)  J : ARRAY_INDEX  
4)  var S : LARGE_INTEGER  
)

Edit? delete

Delete which element? 3

Edit? $
$
$
(  
1)  A : SMALL_INTEGER_ARRAY  
2)  I : ARRAY_INDEX  
3)  var S : LARGE_INTEGER  
)

Edit? grammar

<PROCEDURE DATA PARAMETERS> ::= ( <PROCEDURE DATA PARAM>* )

Edit? up

Edit? up

Edit? down 2

Edit? $  
Begin  
1)  exit ... ;  
2)  { <CONCRETE EXTERNAL SPECIFICATIONS> }  
3)  var K : INT := I;  
4)  { <KEEP SPECIFICATION> }  
5)  S := ... ; ...  
6)  end

Edit? down 5

Edit? show

S := 0;  
loop   
   if I > J
then leave
else assert  \( S = \text{ARRAY\_SUMMATION}(A, I, K) \)
& \( K \in [I..J] \);
\( S := S + A(K) \);
\( K := K + 1 \)
end
end

Edit? quick
1)  \( S := ... \)
2)  loop ...

Edit? down 2

Edit? $\$
Loop
1)  if ...
2)  end

Edit? down 1

Edit? $\$
1)  if ...

Edit? down 1

Edit? $\$
If
1)  \( I > ... \) then
2)  leave
3)  \{ <\text{ELIF PART LIST}> \} Else
4)  assert ... ; ...
5)  end

Edit? down 4

Edit? $\$
1)  assert ...
2)  \( S := ... \)
3)  \( K := ... \)

Edit? show

assert \( S = \text{ARRAY\_SUMMATION}(A, I, K) \) \& \( K \in [I..J] \);
\( S := S + A(K) \);
\( K := K + 1 \)

Edit? save
Save which element? 2

What name to save it under? switch

**** Element number 2, a <STATEMENT>,
has been saved with name SWITCH

Edit? workspace

Which saved element? ?

Commands are: <saved-name>, ALL, NAMES, ^E, ?, ??.

Which saved element? all

Element SWITCH is a <STATEMENT>:
S := S + A(K)

Edit? delete

Delete which element? 2

Edit? $
1) assert ...
2) K := ...

Edit? insert

Using what? ?

Commands are: Parser, Workspace, ^E, ?, ??.

Using what? w

Insert after which element? 2

Which element (by name)? switch

Edit? $
1) assert ...
2) K := ...
3) S := ...

Edit? show

assert S = ARRAY_SUMMATION(A, I, K) & K in [I..J];
K := K + 1;
S := S + A(K)

Edit? undo
**** Going UP - all edits done after going DOWN are ignored

Edit? down 4

Edit? $
1)  \text{assert ...}
2)  S := ...
3)  K := ...

Edit? down 2

Edit? $
1)  S :=
2)  S + ...

Edit? down 2

Edit? $
1)  S
2)  +
3)  A ...

Edit? replace

Using what? parser

Replace which element? 1

Parser - end input with ^Z
S-1
^Z

Edit? show

(S - 1) + A(K)

Edit? $
1)  S - ...
2)  +
3)  A ...

Edit? exit

Finished editing COMPUTE_SUM
CHAPTER TWO

REVIEW OF CURRENT SYSTEMS

There now exist a few structure editors, most notably LISP editors and PASCAL editors. Structure editors are beginning to be used and considered in sufficient numbers to cause debate about the relative merits of text and structure editors.

2.1 Text Editors Versus Structure Editors

In any programming environment there exists the difficult choice between a text-based editor and a structure editor or a reasonable combination of the two. Although programming environments are beginning to become more numerous, the only programming environment in widespread use has been the LISP programming environment.

The two most widely used LISP systems have been INTERLISP and MACLISP. The INTERLISP editor is a structure editor (see Section 2.2) and the MACLISP editor is a text-based editor. The general controversy of text editors versus structure editors has been carried on in the environment of LISP editors between the users of INTERLISP and the users of MACLISP.
A recent article by Erik Sandewall [Sandewall 78a,b] in Computing Surveys has brought part of this debate into print. Sandewall describes the programming environment of LISP in his article, and specifically describes the INTERLISP and MACLISP systems. This article describes the various aspects of the LISP programming environment much like this report describes the Gypsy verification environment.

The benefits and drawbacks of the INTERLISP structure editor and the text-based MACLISP editor are discussed in Sandewall's article. Three dimensions of editing capabilities are proposed and the choices taken in each case by INTERLISP and MACLISP are described. The first dimension is the type of object on which the editor operates. The INTERLISP editor operates on the internal list structure and the MACLISP editor operates on text. In this dimension, the internal list structure is the preferred object of editing, according to the author of this article, because the internal structure is the real structure of interest rather than the incidental textual representation. The second dimension concerns the user interface. The INTERLISP editor uses a "hard copy" interface in that the current state of the item being edited is shown through a "print" command. This method is suitable for a hard copy (paper) terminal. The MACLISP editor uses a dynamic screen
editor which maintains the current state of the edited item on the screen for the user at all times. In this dimension, the preferred choice is the dynamic screen user interface because of its ease of use. The third and last dimension of editing capabilities concerns the size of the incremental editing capability. The INTERLISP editor can edit on any LISP expression and the editing commands are resident in the system. The MACLISP editor modifies the original source file and reloads the file into the programming system. In this dimension, the preferred choice is the resident INTERLISP editor with its arbitrarily small incremental capabilities. Using Sandewall's criteria and opinions on which method is best, the INTERLISP editor is preferred in two out of the three dimensions.

Richard Stallman of the MIT Artificial Intelligence Laboratory where MACLISP originated, has responded to this article in a later edition of Computing Surveys [Stallman 78]. He defends the text-based MACLISP editor with a list of advantages to editing text rather than list structure. Briefly, the advantages cited are:

1. The form and style of the program and its comments are maintained in a text file;

2. Operations involving syntactically incorrect text can be performed;
3. The text editor can provide commands with some understanding of the syntactic structure;

4. The text editor can more easily support dynamically extended syntax;

5. The text editor can be used for other languages;

6. A residential list structure editor can introduce semantic errors which can destroy the LISP environment;

7. The screen-oriented editing is superior.

In the LISP environment it is obviously more difficult to choose between a text-based and a structure editor. Some of the points in question are more easily decided upon in the case of the Gypsy Verification Environment. Given the three dimensions of editing described by Dr. Sandewall, the Gypsy Structure Editor uses the same choices as does INTERLISP. The Gypsy editor edits internal structure, has a "hard copy" user interface, and is a resident editor which allows editing at any increment. As with the INTERLISP editor, the only bad choice is the "hard copy" user interface. The Gypsy editor was designed with this type of interface because the hardware environment of the user interface is not defined. It must be possible for
a user to be able to use the editor while on a hard-copy terminal. It would be very interesting to develop an optional screen oriented user interface, but this would be a very difficult task.

In relation to the stated advantages of editing text over internal structure in LISP, some of the advantages become disadvantages in the Gypsy System. The reloading of LISP text into the LISP system is not very expensive because of the simplicity of LISP syntax. The Gypsy syntax is much more complex and hence, more expensive to parse. The analysis performed in the Gypsy System makes incremental development critical and the expense of reparsing much higher.

Some of Dr. Stallman's points are not applicable to the Gypsy System, like dynamic syntax extension and the possible semantic bugs introduced through the editing process.

2.2 The Interlisp Editor

The INTERLISP editor [Teitelman 75] is a structure editor built into the LISP environment. Any LISP s-expression can be edited, either functions or data. During editing, changes which would result in an illegal LISP expression are not allowed, hence the syntactic
correctness of the expression is maintained at all times. Also, the LISP editor is focused on some subexpression of the structure being edited. The current focus is displayed with various print commands and changed with attention changing commands. Elements of the subexpression can be replaced or deleted and new entries can be inserted into the expression. The INTERLISP editor has additional features, including undo, macros, and access to the general power of LISP from inside the editor.

The INTERLISP editor displays the expression in full and shortened forms. The "P" command shows the current expression to a depth of two and the "PP" and "?" commands show the current expression in its full form, the first being pretty printed, the second not.

The attention changing commands direct the editor around the internal LISP tree structure. A number directs the editor to change the current focus to the numbered subexpression. The expression retrieved is equivalent to that gotten with the LISP NTH function with the number as the argument. The number 0 is used to direct movement upward one level in the expression, to the expression containing the current expression. Movement can also be directed with a find command.
Modification of the current expression can be done with replacement, deletion, and insertion commands. Any of the subexpressions can be replaced by number with a new LISP expression. Since all input (including commands) is parsed using the standard LISP parser, the syntax of the new replacement is guaranteed to be correct. A subexpression can be deleted by replacing it with nothing. Insertion of a new element is done by inserting it before one of the existing elements or by inserting it at the end.

Other modification commands include replacement of one expression for another everywhere in the current expression. Because of the need to preserve the LISP syntax, the modification of any parentheses in the expression pose special problems. The parentheses are not really a part of the expression, but are displayed as an indication of the underlying tree structure. Hence, changes in the parentheses are really changes in the tree structure. The parenthesis changing commands are the most difficult INTERLISP editor commands to understand and master.

The INTERLISP editor provides a few very powerful editing commands. The UNDO feature is particularly powerful not only because it can undo the last command or the last few commands, but because it can undo them selectively. This means, for example, that the effects of the third most recent command can be reversed without reversing the two
edits done after it. Since the INTERLISP editor is built into the LISP system, all of the power of LISP is available to the user at all times.

2.3 A General Program Editor

The need for a general, language independent programming environment was realized by Martin Yonke [Yonke 75] several years ago. He designed and partially implemented a programming environment consisting of a parser and editor for construction and modification of programs. This system works off of a formal description of the syntax and some of the semantics of a particular language.

This system requires an "expert" to describe in the formalism the programming language to be manipulated by the system. This description includes the complete syntactic definition of the language and some limited semantic information about the use of names in the language.

Once the language has been formally described, the two tools of the system use this description as a guide to the understanding and manipulation of programs written in this language. The system has the advantage of having every component being driven by the same formal description of the language.
The formalism used in Yonke's system to describe the syntax is basically an augmented BNF form. There are two major production types, the ordered sequence of terminals and non-terminals (referred to in the Gypsy editor as a "record" production), and an alternating or repeating sequence of objects. Yonke has added one other production which is a bracketed alternating or repeating sequence. The effect of the addition of this production is to reduce the number of non-terminals required to describe certain kinds of languages.

The internal structure of the parsed program is an n-ary parse tree. All terminal symbols in the original text are saved in the internal representation thereby simplifying the unparsing or printing of the program back to the user. Included in each level of this tree are pointers to the corresponding production in the language definition. This information is used for interaction with the parser.

There is a strong interaction between the parser and editor components, along with an error recovery mechanism, which greatly increases the flexibility of the user interface. When an error occurs during construction or modification of a program, the system tries to recover from the error by using the parser's look ahead symbol set and a spelling corrector. If the system cannot recover from the error on its own, the user is notified and is allowed to use
the editor to correct the error.

Although this system was presented as an example of a general, language independent program environment, it suffers from a lack of experience with a variety of program tools and a variety of languages. This system is greatly tailored toward the two tools implemented and the one language, PASCAL, to which it has been applied.

For the purposes of program construction, modification, and unparsing, the n-ary parse tree with all of the original terminal symbols is quite powerful. But it is not general enough for various program analysis tools. The lack of an internal canonical form for equivalent syntactic constructs would make program analysis unnecessarily complex and difficult.

The formalism used to define the syntax of the language is far from language independent. It is a simple context free grammar description with special productions (such as the bracketed sequence) which are useful only for PASCAL and similar languages. The semantic information used by the system is only simple scope and definition rules for the use of names in the language. This is only useful for languages such as PASCAL.

Although the Gypsy editor was written to handle only the Gypsy language, it contains at least as much generality
as Yonke's system. Unlike Yonke's system, the Gypsy editor is constrained to work efficiently and usefully in a large diverse programming environment.

2.4 A Pascal Programming Environment

A first step toward the construction of a programming environment is described by Huet et al. [Huet 75]. This environment supports the creation and modification of PASCAL programs. The program editor is considered to be the representative component of a varied and powerful programming environment.

The PASCAL language is formally described in the system in a production language which is very similar to the one used in the Gypsy editor. There are two kinds of productions, fixed arity productions and varyadic productions. The fixed arity productions are a fixed sequence of terminal and non-terminal symbols (similar to "record" productions in the Gypsy editor) and the varyadic productions have an arbitrary number of symbols.

The PASCAL program is parsed into a tree which is similar to the tree describing the syntax. Attached to any node can be a comment, although no mechanism is given for editing the comment. This internal form is used by the parser, editor, and unparsers. The parser uses this tree to
direct the parsing of new PASCAL segments for inclusion into the structure. The editor uses this structure for searching and modifying the program. And the unparsr prints the terminal symbols of the tree (corresponding to the leaves of the tree) according to the structure of the tree.

The search facility described is fairly powerful. The simpler search looks for only a leaf of the tree (a symbol) or a node of the grammar (such as an "if statement"). The more complex search performs general tree matching. Problems arise when the user does not understand the internal representation of the program text and looks for a sub-tree which doesn't exist as such. For example, the expression "A + B" is not a subexpression of "A + B * C" given the normal precedence rules.

The user interface of Huet's system benefits from two different features. Information is output to the user via a high speed, cursor controlled screen. Part of the screen is reserved for diagnostic messages and user commands, and the rest is left for the display of the program text. The command scanner is controlled by the same system which controls the creation and modification of PASCAL programs. This allows an internal structure representing the command to be built and modified by the parser and the editor. The internal command structure is then interpreted by the system much like a PASCAL program
might be interpreted by the system.

Huet's system suffers from one of the deficiencies of Yonke's system (see Section 2.3). The internal representation of the program was chosen to facilitate parsing, editing, and unparsing. It is not necessarily the best representation for the general program analysis which might be desired in the programming environment.

2.5 A Program Constructor

Nico Habermann and Raul Medina-Mora [Haberman 79a,b] use a language independent editor in their program and system development environment. This system supports the incremental development of programs and systems with a software management system.

To aid in the gradual construction of a system, this programming environment separates the program construction process into a series of easily controlled steps. For example, the compilation of a program is separated into the lexical, syntactic, semantic, code generation, and module linking steps. The user is involved in each of these steps, thereby reducing the work resulting from errors in any phase.
The emphasis on the editor is as a program constructor. The user constructs or modifies the program using program constructs or templates. The editor uses a language description to guide the editing process and to guarantee the syntactic correctness of the program. At present, no semantic analysis is done during this construction or modification phase.

The user constructs a program using the editor by filling in templates with language constructs. If taken down to the expression level, the user would have to construct the expression in a prepolish form.

The language description used by the editor includes all of the constructs of the language and specifications of where the constructs are legal. The syntax of the language is thereby defined. The language constructs are "operators" for the editor. There are terminal and non-terminal operators with the non-terminal operators consisting of fields to be filled in. Associated with every construct is an unparsing scheme which is used to print the program segments.
CHAPTER THREE

THE GYPSY VERIFICATION ENVIRONMENT

3.1 The Gypsy Language And Its Internal Form

Gypsy is a PASCAL-like language (see Appendix A) designed to support specification and verification in the development of systems software. For the purposes of verification, Gypsy has no global variables hence it does not have the hierarchical block program structure of PASCAL. The global definitions lie in an essentially flat name space with access control governed by the explicit naming of those global entities allowed access.

The global entities which are definable in Gypsy are called "units" and are the routines, functions and procedures; the user type definitions called type units; the global constant data, called const units; and the lemma units, which are used as specifications and also used at proof time as a means of factoring the proof.

Beside the function, procedure, type, const, and lemma units, there exist "scopes" which contain groupings of these units into name spaces. This is used primarily to reduce the size of the name space as seen by any particular
unit for convenience to the programmers. Such scopes can also be used for access control.

Any Gypsy program can contain either executable code, specifications, or both. A program which contains no specifications is trivially verifiable and is a normal algorithmic program. A program which contains no executable code is essentially a mathematical theorem which may be proven if desired but which will do nothing if executed. A program which contains both executable code and specifications can be both executed and verified. Verification consists of proving mathematically that the specifications and the code are equivalent. Therefore, the power of the verification process depends upon the strength of the specifications.

Every Gypsy program introduced into the Gypsy Verification Environment is translated into an internal form and stored as LISP data. The form is a prefix form similar to LISP functions and has some relation to the syntax tree representation of the program. Expressions are stored in a representation which is very close to the syntax tree but units and statements are stored in a table form to aid in component lookup during processing. An example of the Gypsy form and internal prefix form is an assignment statement. The Gypsy form is:
A := B + 1

The internal prefix form for this statement is:

(ASSIGN A (PLUS B 1))

It is important to note that the internal form was designed well in advance of the creation of the structure editor. The internal form was designed to aid in the processing involved in the verification process and, where needed, the internal form differs significantly from the Gypsy language equivalent. The structure editor had to be designed around the existing internal form, no matter how inconvenient that might become.

3.1.1 The Gypsy Syntactic Translator

The Gypsy parser and translator is a table driven LALR(1) parser. The tables were generated from a BNF-like grammar (see Appendix A) by the BOBSW system as implemented and maintained at the University of Texas by Wilhelm Burger [Burger 74]. The parsing algorithm was also implemented in LISP by Wilhelm Burger. The grammar definition used by the table generator was derived from the syntax diagram definition of the language [Good 78] by the author as were the semantic actions required to do the actual translation from Gypsy to the internal form.
This table-driven parser has the capability of beginning the parse with any production in the grammar. Normally, files are parsed using the topmost production as the starting point, but the editor requires the capability of parsing any Gypsy segment from the user. This BOBSW parser allows this capability in a very straightforward and simple manner.

3.1.2 The Gypsy Infix Printer

Infprint, the pretty print package, allows a segment of the internal form of a Gypsy program to be displayed to the user in the normal Gypsy form. This package, which is used by almost every component of the system, was designed and implemented by the author.

Two features of this pretty printer aid in the editing process. When the current Gypsy segment being viewed is displayed to the user, the editor retrieves the internal segment corresponding to each non-terminal in the appropriate grammar production. This segment is then passed to Infprint for printing. This means that the pretty printer must be able to print, in isolation, every possible internal segment corresponding to a non-terminal in the grammar definition. Also, this segment might be very large, so the printer has the capability of printing to a certain
level and then just indicating that the segment continues at that point. The unprinted segments can be viewed by descending to that level with the editor.

3.2 Program Development, Verification, And Testing

Program development through incremental development is controlled by the toplevel component of the system. This component was designed and implemented in major part by Mark Moriconi [Moriconi 77]. This component also provides the executive user interface and database management.

Verification condition generation is done through the VCGEN component of the system. This component was designed and implemented by Richard Cohen. Upon user direction, this component analyzes a program unit's executable code and specifications and produces a theorem which, if proved, guarantees the equivalence of the specifications and the code.

Theorem proving is done through a natural deduction system called PROVER which is derived and maintained from a system devised by Dr. W. W. Bledsoe [Bledsoe 73,75]. This work was done by Mabry Tyson. PROVER allows the user to direct a proof while doing all of the bookkeeping and proof checking itself. It has powerful automatic deduction capabilities which can be called into use to aid in the
proof.

Testing can be done while inside the programming environment through the interpreter component of the system. This system was written by Judith Merriam. Interpreting can begin with any routine unit in the database using arbitrary data parameters. Tracing facilities allow detailed examination of the interpreting process. Actual program execution on the target machine is achieved through the Gypsy compiler designed by Lawrence Hunter and implemented by him and Lawrence Smith [Hunter 79]. Presently, the Gypsy compiler produces code for the PDP-11 machine where it executes in its own environment.
CHAPTER FOUR

THE GYPSY STRUCTURE EDITOR

The Gypsy Structure Editor is a table-driven syntax directed structure editor. All required information used to accomplish the editing process and all grammar structure information used by the editor is contained in a table which consists of both data and procedural entries. The editor is syntax directed through its use of the table and in its use of the system parser for user input. The editor modifies the structural internal form of the program which is maintained by the verification environment.

4.1 The Syntax Directed Editor

The Gypsy segment being edited is viewed as a grammar syntax tree with the appropriate production as the starting point. Grammar productions have two forms, either a "record" form or a "repeating" form. Input from and output to the user is directed by the syntax of the Gypsy segment involved.
4.1.1 Program Structure As Grammar Tree

When a program is parsed, the syntax tree corresponding to that program is computed. The form of the program (but not necessarily the semantics) is determined by the grammar definition and the set of syntax trees which it allows. Editing this program is solely concerned with the modification of the form of the program and hence must be completely aware of the relationship of that program to its syntax tree.

The editor views the structure of the program in the same way in which it represents the language grammar in its internal table (see Section 4.4). This structure is used to direct movement through the program and to direct modifications as desired by the user.

4.1.2 Program Grammar Structure

The grammar definition used by the table driven parser (but not the editor) is a simplified BNF form. A grammar production is normally defined as a sequence of terminals and non-terminals. As an example, an expression might be defined as:

\[ <\text{EXPRESSION}> ::= <\text{VALUE}> \ <\text{BINARY \ OP}> \ <\text{VALUE}> \]

A grammar non-terminal is allowed to have more than one
possible definition and one of those definitions is allowed to be the "EMPTY" production. This corresponds to an optional component of a Gypsy segment.

A production can be either left or right recursive (but not both) in order to create repeating elements. Thus a grammar production can be either a non-recursive list of terminals and non-terminals, a recursive definition, or the empty production.

4.1.3 Program Grammar Elements

The structure of the grammar used by the editor is very similar to the parser grammar described above. The editor table definition distinguishes between two different kinds of grammar productions, the "record" production and the "repeating" production. The EMPTY production (which can appear as an alternate to either a record or repeating production) is handled by designating the element as optional or deletable.

4.1.3.1 The Record Grammar Definition

The record grammar definition is composed of a fixed number of terminals and non-terminals (non-recursive). As an example, the definition of the send statement is as
follows:

\[
\text{SEND STATEMENT} ::= \text{SEND} \text{<EXPRESSION>} \text{ TO} \text{<VARIABLE REF>}
\]

This definition consists of two terminal symbols (SEND and TO) and two non-terminal symbols (<EXPRESSION> and <VARIABLE REF>). An optional element of the production is indicated by marking that element as capable of being deleted.

For the purpose of display and user interaction, each record grammar element is retrieved and stored separately. When displayed, an empty element is shown as a comment giving the name of the non-terminal whose corresponding program realization is missing.

4.1.3.2 The Repeating Grammar Definition

A repeating grammar definition consists of a single non-terminal and an indication that the element may be repeated. Whether or not the number of elements can shrink to zero depends on whether or not the production allows the empty symbol as a definition (an optional element). None of the repeating grammar definitions in the Gypsy language have an upper bound on the number of elements allowed. Nevertheless, the editor contains a mechanism to restrict the number of elements allowed to be added to a repeating Gypsy segment. An example of this kind of production is the
routine parameter list:

<PARAMETER LIST> ::= ( <PARAMETER>* )

The repeating elements in the actual Gypsy segment corresponding to the production are stored in a list. This is a dynamic list which grows and shrinks at the user's command. The order of the elements is also unimportant and may be modified at will.

4.1.4 Syntax Directed User Interaction

The editor has a grammar structure view of the program and the editor presents the same view to the user. When displaying the current program segment, the parts corresponding to each non-terminal in the associated production are presented to the user separately and numbered. The various segments displayed can be referenced in other commands by using the numbers displayed with the segments. An example of this command, the QUICK command (also abbreviated by the escape or altmode character) is the following:

   Edit? quick
       send
       1) A + 1 to
       2) DATABUFFER

   Edit?

The current program segment can also be shown in its normal
Gypsy form with the SHOW command (see Section 1.4).

Attention is directed toward different parts of the program by moving through the grammar tree and changing the segment under consideration (called the "focus"). Movements are directed up or down the grammar tree. A downward movement, using the DOWN command, is done by choosing one of the non-terminals of the current production and making the production of that non-terminal the current production. The program segment corresponding to that non-terminal is retrieved and made the current focus. An UP command reverses the effects of the DOWN command and returns to the former production and program segment. The effects of the DOWN and UP commands are shown in the following examples:

```
Edit? SHOW
send A + 1 to DATABUFFER
Edit? DOWN 1
Edit? SHOW
A + 1
Edit? UP
Edit? SHOW
send A + 1 to DATABUFFER
Edit?
```

Input from the user is syntax directed since it is parsed and translated by the system parser. The context of the modification being done directs the parser to the
appropriate production. Any user input to a program segment is directed toward some non-terminal in the current grammar production and the production associated with that non-terminal is used to direct the parser syntactically.

The editor can also display the grammar production associated with the current focus. This information is not needed by the user to perform any editing task, but it might be useful. An example of the GRAMMAR command is the following:

```
Edit? GRAMMAR

<SEND STATEMENT> ::= SEND <EXPRESSION> TO <VARIABLE REF>

Edit?
```

4.2 The Structure Editor

The Gypsy Structure Editor edits the structured internal form maintained by the programming environment. The editor maintains a structured grammar view of the program but this view of the program often does not agree with the actual internal form being modified. Therefore, procedural information is stored with every grammar entry in the editor table (see Section 4.4.3) which allows the editor to perform the necessary modifications on the internal form.
The editor presents the structured form to the user through the display package and the command structure. Gypsy input from the user is accepted as structured input by sending it through the parser. Commands allow the user to make modifications to the Gypsy segment in a structured way and the editor must translate these directives into the appropriate modifications to the internal form.

4.3 The Structure Editing Functions

The editing functions implemented form a small set of primitive indivisible modification functions. These functions form a complete set in the sense that any desired legal modification can be realized as a sequence of these editing functions applied to the internal form. Each of these primitive functions preserve the syntactic validity of the internal form therefore guaranteeing that any allowed combination of these editing functions will also preserve this integrity.

4.3.1 Functional Modifications and Syntax

In order to insure consistency and correctness, the internal form being modified should always be kept in a legal, correct form as defined for that internal form. The internal form defined for the Gypsy Verification environment
does not have any allowed states which correspond to illegal syntax in the original Gypsy text. Therefore the Gypsy program must be kept in a syntactically correct form to maintain the consistency and validity of the internal form. Since there exists an internal form for every correct syntactic form of a Gypsy program, maintaining syntactic correctness is both a necessary and sufficient condition for ensuring a correct internal form.

Each primitive editing command is indivisible, hence the primitive command need not insure that the correct internal form is maintained throughout the modification, only that if the form was correct before the modification was started, then the form is still correct after the modification has been completed. This restriction on the commands necessarily makes them incapable of performing some modifications simply. For example, deleting a semi-colon from one place and inserting it at another requires the program to be syntactically incorrect for some period of time which is not allowed. The editor must provide sufficiently powerful commands built up from these primitive commands in order to give the user the flexibility needed to perform these kinds of modifications.
4.3.2 Editing Functions and Grammar Elements

The editing commands and their realization as sequences of primitive editing functions are directed toward the program segment currently viewed by the user. This display is presented as a production consisting of numbered non-terminal elements. The only objects available for modification are these non-terminals using as a reference the corresponding number of each non-terminal.

Each primitive command takes as argument one of these non-terminals (or the number associated with that non-terminal element) which it then manipulates in isolation of the other elements. The context free characteristic of the grammar allow this factoring of the functions and their modifications.

The important information which allows the editor to correctly manipulate these non-terminals is the relationship among these non-terminals. Certainly two non-terminal elements are equivalent if they have the same name and hence the same associated production. Also two non-terminals can be compared if one is a subset of the other. For example, the \texttt{<SEND STATEMENT>} is a subset of the class of \texttt{<STATEMENT>} and therefore a \texttt{<SEND STATEMENT>} can be used where a \texttt{<STATEMENT>} non-terminal is required. The relation does not hold the other way, however, since a \texttt{<STATEMENT>} might not
necessarily be legal where a <SEND STATEMENT> is.

Normally one non-terminal under consideration comes from the current production being viewed and the other is indicated by the user as a possible replacement. If the other non-terminal element comes directly from the user through the parser, then the parser will accept a Gypsy segment of the most general subset kind, thereby insuring their compatibility. If the Gypsy segment comes from another part of the program, then it already has a non-terminal label associated with it. For certain operations to be allowed, the non-terminals involved must either be equivalent or the proper subset relation must hold. These subset relations are derived from the grammar table structure itself dynamically by the editor. Therefore modifications can be made to the editor's grammar without concern for the effect on these subset relations.

4.3.3 The Primitive Editing commands

The following three primitive editing commands form a complete set of the necessary editing commands needed to perform any desired modification to the program. Each primitive command, when applied to a valid internal form, returns another valid internal form with the appropriate modification completed. The three primitive editing
commands are REPLACE, DELETE, and INSERT.

4.3.3.1 The Replace Editing command

The REPLACE primitive command removes an entry in the current focus and replaces it with a compatible entry. The replace operation is always a valid operation on any non-terminal element. Compatibility between the old entry and its replacement entry is determined by their associated non-terminal symbols and the conditions described above. If the non-terminals are equivalent then the replacement is valid. Otherwise some subset relation must hold and the relation must be that the non-terminal associated with the old entry is a member of the set of the non-terminals associated with the replacement entry. Since the parser always accepts the most general object from the user this relationship holds for any valid input from the user. Using the example described above, if a <SEND STATEMENT> appears as part of a <STATEMENT LIST>, then the most general form is a <STATEMENT>. Hence, if the send statement is replaced by the user, it need not be replaced by another send statement, but by any general statement since the subset relation between the <SEND STATEMENT> and the <STATEMENT> holds in this case. However, if the send statement is a label on an await statement, then the most general form is a <BUFFER STATEMENT> (see appendix A), and the parser will not allow a
more general statement to be parsed in since the subset relation does not hold in this case. As is indicated in this example, the subset relation is quite dynamic and depends upon the context of the table entry and the grammar tree of the program being edited.

The REPLACE command differs slightly in its effect on the record type of grammar production and the repeating type. In the case of the repeating type, the element of the list is simply replaced, unaffected by the rest of the list or the size of the list. An example of this kind of replacement is the following:

```
Edit? quick
( 1) A : INTEGER
 2) B : BOOLEAN
)
Edit? replace parser 1
Parser - end input with ^Z
C : character
^Z
Edit? SHOW

(C : CHARACTER; B : BOOLEAN)
```

In the case of the record type of grammar production, the new element is stored in place of the old element. Here it is important to use a replace command instead of performing a delete and then an insert because many elements of productions are not optional and their
deletion would generate a syntactically incorrect program which is not allowed. In fact, a record type of grammar production does not grow and shrink in size as a repeating type does during execution of delete and insert commands, but remains a fixed length. Deleted elements are still represented in the focus but are considered "empty". Hence an insertion into an empty place is considered as a replacement of the empty entry by something else. It is even possible to delete an entry by the action of replacing it with nothing. An example of the REPLACE command follows:

```
Edit? quick
   send
1) A + 1 to
2) DATABUFFER

Edit? replace parser 1
Parser - end input with ^Z
a-1
^Z
Edit? show

send A - 1 to DATABUFFER
Edit?
```

4.3.3.2 The Delete Editing command

The DELETE primitive command either removes the element from the list of repeating elements or replaces the record element with the empty element. The DELETE command is not always a valid command. If deleting a repeating element causes the entire list to become empty then the
effect is to delete the entire production. This is only allowed if the parent production is deletable. For example, a procedure parameter list is made up of a list of procedure parameters. Deleting the last parameter would have the effect of deleting the entire parameter list which is not allowed. Therefore the deletion of the last parameter could not be allowed by the editor. An example of a deletion of a repeating element is:

```
Edit? quick
  (1) A : INTEGER
  2) B : BOOLEAN
)

Edit? delete 1
Edit? show

(B : BOOLEAN)

Edit?
```

Deleting a record type entry consists solely of checking to see if the non-terminal associated with the entry is an optional one. If so, the entry is removed and the proper indication is stored to indicate that the entry is now empty.
4.3.3.3 The Insert Editing command

The INSERT primitive command adds a new entry to a list of entries. This command is valid only for the repeating type of grammar production since the recorded type is a fixed set of entries. The current Gypsy grammar has no restrictions on the size of any of the repeating productions in the grammar, but the mechanism exists to test the size and impose restrictions.

The list of repeating elements expands dynamically with insertions and shrinks with deletions. In this case a replacement is equivalent to an insertion followed by a deletion or vice versa. An example of an insertion is the following:

```
Edit? quick
  (
  1) A : INTEGER
  2) B : BOOLEAN
  )

Edit? insert parser 2
Parser - end input with ^Z
  c : character^
  ^Z
Edit? show

(A : INTEGER; B : BOOLEAN; C : CHARACTER)
```

Edit?
4.4 The Table Directed Editing Algorithm

All language specific details of the editing process are isolated in the editor table. This includes the grammar definition used by the editor to understand the structure of the program and the procedural entries used to accomplish the actual modifications. A claim is made that this makes the editor language independent to a limited extent.

4.4.1 The Table and Language Independence

Since the grammar definition is kept in a separate table, along with the routines required to modify the internal form, minor changes to the Gypsy language definition are easily handled by modifying the editor's table definition, and should require no changes to the editor itself. Thus the editor is fairly independent of minor fluctuations of the given grammar. The degree of independence of the editor over the set of all language definitions depends on the assumptions regarding the grammar definition which have been built into the editor. This structure editor was designed around the Gypsy language and no real attempt was made to design a general structure editor for all languages. The editor was designed to be independent of minor modifications in the Gypsy language to facilitate upgrading. A claim is made that this editor is
applicable to the class of block structured languages to which Gypsy belongs. This set includes the computer languages ALGOL and PASCAL.

Several assumptions have been built into the Gypsy Structure Editor regarding the language being edited. In order to allow modifications in one part of a Gypsy segment to proceed independently of other segments, the language must have a grammar definition which is relatively context free. Any LR(1) grammar is sufficient. The grammar used by the editor must be in the form described above which uses the empty symbol to denote optional items and factors productions into those which have repeating elements and those which do not. No analysis has been done to define the set of languages allowed by these restrictions.

The Gypsy Structure Editor must maintain the syntactic correctness of the program at all times, but this restriction comes only from the programming environment. The editor algorithm does not distinguish between syntactically correct and incorrect programs. The only requirement is that the editor be able to modify the internal form corresponding to any allowed modification.

This editor is best suited toward block structured languages such as Gypsy and PASCAL. Line oriented non-structure languages such as APL and FORTRAN could be
handled by this editor (with a suitable grammar definition) but their lack of structure would degenerate the editor to a line oriented token editor. A highly structured language such as LISP could also be handled but the grammar definition would be so trivial as to reduce the obvious power of the editor with respect to the block structure languages. This editor would become a subset of the INTERLISP editor as described in section 2.2.

The language independence of the algorithm used by the Gypsy Structure Editor has been claimed for a reasonable subset of the current computer languages available. No attempt has been made to prove this assertion through formal arguments or through implementation.

4.4.2 The Table Structure

The editor table is used to direct the editor algorithm through all required operations. The editor's grammar definition contains all of the information needed by the editor regarding the structure of the language. The grammar is structured in a form similar to that used by the parser and the editor table has been built around and incorporates the grammar definition. The table is structured as a set of entries which refer to each other in a way which corresponds to the structure of the grammar.
Each table entry has a set of elements which are used by the editor algorithm to display and modify the Gypsy segment corresponding to the table entry. These elements include the name of the table entry, the production elements of the grammar definition, the predicates, and the procedural entries.

Of these elements, the grammar definition is represented by the production elements. These production elements point to other table entries thereby representing the structure of the grammar in the table.

4.4.3 The Table Entries

The table entries correspond to the productions of the grammar. Associated with each grammar production is information regarding the display and modification of the Gypsy segment corresponding to the production. The name of the table entry is the non-terminal grammar symbol associated with the production. Wherever possible, the name of the table entry (and the grammar non-terminal) is the same as that used in the grammar definition of Gypsy in the Report on the Language Gypsy: Version 2.0 [Good, 78].
4.4.3.1 The Grammar Production Elements

Each table entry whose associated non-terminal has a structure (\texttt{<IDENTIFIER>} is considered to be a primitive, for example, and has no structure.) contains a sequence of terminals and non-terminals of the grammar production.

The terminal symbols of the grammar are represented as strings in the table entry. They are used only to provide context while displaying the current Gypsy segment to the user. As an example, the procedure header production is defined as the terminal symbol "Procedure", the unstructured non-terminal \texttt{<IDENTIFIER>}, and the structured non-terminal \texttt{<PROCEDURE PARAMETER LIST>}. When displayed, the terminal symbol "Procedure" is printed to the user.

The non-terminal symbols represent the productions of the editor's grammar and name the table entries which correspond to the grammar productions. The correlations between the non-terminals and the parts of the Gypsy segment are defined by the elements of the table entry for each non-terminal.
4.4.3.2 The Primitive Functions

The primitive editing functions form a complete set in that any legal modification of the Gypsy program can be performed as a sequence of applications of the editing functions. The functions are considered primitive because more elementary operations are not needed and sometimes don't make any sense. The primitive editing functions of the Gypsy Structure Editor are retrieve, store, and remove.

The retrieve function is used to get a part of the current Gypsy segment. Each non-terminal symbol is contained in one or more grammar productions, and therefore the Gypsy segment corresponding to this non-terminal is contained in one or more different kinds of Gypsy program segments. The retrieve function goes into the current internal form and extracts the program segment corresponding to the non-terminal. The retrieve function acts in a different way on the record and repeating grammar forms. A Gypsy segment corresponding to a record grammar definition is quite varied in form. Each non-terminal of the record grammar definition has associated with it a retrieve function which can extract the appropriate Gypsy segment.

The repeating grammar form consists of one non-terminal which is repeated. The internal form associated with the repeating grammar definitions in the
Gypsy System are always represented as a list of objects. Because of this, the retrieve function for the repeating form is always a simple function which extracts the nth object in the list. Because of this simplification, this retrieve function has been omitted from the current editor table.

The store function performs the opposite of the retrieve function as it puts a Gypsy part into the current Gypsy segment being viewed. The store function takes a non-terminal symbol, an associated Gypsy internal form segment, and the current Gypsy focus and modifies the current focus by replacing the part of the focus corresponding to the non-terminal with the new Gypsy segment.

As with the retrieve function, the store function proceeds differently depending on whether the grammar form is a record or a repeating form. The record grammar form consists of a set of non-terminals and associated with each of these non-terminals is a specific store function. A Gypsy segment is stored by applying the appropriate store function to the Gypsy segment and the current Gypsy focus. Since the form of a repeating Gypsy segment is represented as a list, storing is done by expanding the list and adding a new entry.
The remove function allows deletion by removing Gypsy segments from the program. The remove function takes a non-terminal and a current Gypsy focus and modifies the Gypsy focus in such a way so that the program segment corresponding to the non-terminal is either removed or replaced by an indication that the form is empty.

Every non-terminal of a record grammar production which can be deleted has associated with it a remove function. This remove function replaces the associated Gypsy segment with the entry used to indicate that the optional part is missing or empty. The representation of a repeating form is a list of objects and the remove function acts on this form by shrinking the list and deleting one of the elements.

4.4.3.3 The Primitive Function Predicates

The actions of the primitive functions on the Gypsy program are constrained by the syntactic definition. These constraints are represented in the editor table by the primitive function predicates. The editor also adds constraints in order to allow editing changes which are not strictly single modifications according to the grammar.

The syntactic constraints for the Gypsy programming language consist only of deletion constraints although
another language might have replacement and insertion constraints. Only those Gypsy non-terminals which are considered optional or have a definition which includes "empty" can be deleted for record grammar productions. The last element of a repeating grammar production can only be deleted if the entire item is optional.

The Gypsy editor includes additional or modified grammar productions and additional constraints in order to allow editing changes by the user which seem natural but would be difficult under the normal grammar and editing algorithm.

The language features which have been so modified are ones which include contextual information in the language by explicitly enumerating the possible syntactic combinations.

An example of this in the Gypsy language is the <TYPE REFERENCE> production. A type reference may be a type name, a scalar subrange, or both. The semantics of this construct is that a type reference is a type name optionally restricted by a subrange, but that the type name may be omitted when the range is given since the type can be derived from the range. In this sense, the type name of a type reference is optional if the optional range is present. This is realized in the grammar by the following production:
<TYPE REFERENCE> ::= <TYPE NAME> 
                <TYPE NAME> <RANGE> 
                <RANGE>

In this production, all of the possible combinations have been included and the <TYPE NAME> and <RANGE> non-terminals have not been marked as optional (and therefore cannot be deleted). If a program includes a type reference which contains both a type name and a range, then normally that type reference cannot be changed to one containing only the range simply by deleting the type name. Although this modification seems obvious to the user, the editor cannot allow it because the type name is not technically optional.

The editor allows this kind of modification by marking the type name as optional and adding a primitive function predicate which checks for the existence of the range before allowing the deletion. In this way the editor allows modifications which agree with the user's view of the language definition but not with the actual grammar definition of the language. This problem reveals a fault with the representation of the language in the grammar and not with the editor itself.
4.4.3.4 The User Interactive Functions

The interaction between the user and the editor consists of input from the user and output to the user. Gypsy segments are accepted from the user using the Gypsy syntactic parser and are printed to the user using the Gypsy pretty print package.

The editor table for each entry has all the information required to parse in a Gypsy segment corresponding to the grammar production of the non-terminal symbol in the table. The Gypsy syntactic parser (see Section 3.1.1) needs only an entry point corresponding to one of the productions of the Gypsy grammar used by the parser. These entry points are kept in the editor table. The internal form prefix returned by the parser might not be in the exact form required by the editor for that entry. A functional entry is included in the table to transform the internal form when necessary.

The Gypsy printer (see Section 3.1.2) was written in a modular form with each Gypsy segment being printed by a function. Entry points to the printer are functional in nature. Each non-terminal of the table has a functional entry point to the printer which also modifies the internal form if required by the Gypsy printer.
4.4.4 The Editing Algorithm

The Gypsy Structure Editor is command driven by the user. Most of the commands available to the user fall into three categories: attention changing commands, display commands, and modification commands.

At any moment, all of the editor commands available are directed toward a certain grammar production of the language and its associated Gypsy segment. In order to modify the various parts of a Gypsy program, the focus of the editor must be changed. This is done with the attention changing commands.

The set of grammar productions is considered to be a tree and movement is directed up and down the tree. The DOWN command takes one of the non-terminals of the current production as an argument and modifies the current focus. The grammar production of the new non-terminal is made the current production and the Gypsy segment corresponding to the non-terminal is made the current Gypsy segment. The previous state is saved and is restored upon execution of the UP command which moves back up the tree. All other movements through the Gypsy program are combinations of UP and DOWN commands.

The display commands take the current Gypsy segment and print it to the user using the Gypsy printer. Each
element of the program part is printed separately with a number to aid in specifying arguments to other commands. Two display commands are provided, the SHOW command displays the current Gypsy segment in its entirety, and the QUICK command displays the segment in an abbreviated way which presents the context without excessive printing.

The basic modification commands are REPLACE, DELETE, and INSERT. The REPLACE command gets a new Gypsy segment from the user using either the parser or the workspace (see Section 4.5.2) and stores that segment into the current focus. The DELETE command checks the delete predicate to make sure that deletion is allowed and then applies the remove function to the appropriate non-terminal. The INSERT command gets a new Gypsy segment and stores it into the repeating list of objects. Note that the INSERT command is only allowed on repeating grammar forms. Other modifications are combinations of these three modification commands.

4.5 The User Interface

Since the Gypsy Structure Editor is a command driven editor, there is considerable interaction between the editor and the user. The usefulness of the editor is directly related to the quality of this interaction in addition to
the power of the editor. The quality of this interaction depends in large part on the view the user has through the editor's user interface.

4.5.1 The Editor Command Language

The editor command language is a keyword oriented language. Each command can be defined to have any number of arguments. The command scanner scans the entire line and accepts any unique abbreviation of the command and its arguments. The arguments not given are prompted for by the command scanner.

The command scanner has two kinds of help features besides the automatic prompting for missing arguments. A HELP command gives a detailed explanation of the effects and use of each command. In place of any command or argument, a "?" may be used. In response, the command scanner lists the commands or arguments which are legal at that point.

Using these features, a user should be able to figure out how to use most of the commands without other documentation. This is important because in order to be useful, the editor should be comfortable to work with. The command scanner (which is used by other system components) was written by Mabry Tyson.
4.5.2 The Workspace

The workspace is an area where the editor allows the user to save Gypsy segments for later use. These Gypsy segments can be used with the REPLACE and INSERT commands for inclusion into the Gypsy program at another place. The difficult task for the editor is to make sure that only legal Gypsy segments get used in the editing commands.

A Gypsy segment can be saved in the workspace with the SAVE command. The SAVE command is directed toward one of the Gypsy segments being displayed. Since the grammar production is present when the save is performed, the appropriate syntactic information about the Gypsy segment can also be saved. An example of the saving and displaying of a Gypsy program segment follows:

```
Edit? quick
  send
1) A + 1 to
2) DATABUFFER

Edit? save 1 sendexp

**** Element number 1, a <EXPRESSION>
  has been saved with the name SENDEXP

Edit? workspace all

Element SENDEXP is a <EXPRESSION>:
A + 1

Edit?
```
This saved information is used to make sure that the saved Gypsy segment is only used in legitimate ways. When input to be used is accepted directly from the user, the parser syntactically checks the Gypsy segment to make sure it is correct for the use directed by the user. There are a variety of problems associated with checking the use of a previously saved Gypsy segment.

The grammar elements (non-terminals) of the language have a partial ordering imposed upon them by the subset relation. As an example, in the Gypsy language, the `<SEND STATEMENT>` is an element of the set of `<BUFFER STATEMENT>`, which in turn is a `<STATEMENT>`. The rule for the use of a saved Gypsy segment for a certain non-terminal is that the grammar element representation of the saved segment must be less than the required non-terminal in the partial ordering relation. Using our previous example, if a SEND statement had been saved in the workspace, it could be used where a `<STATEMENT>` was required because of the relationship previously demonstrated between the `<SEND STATEMENT>` and the `<STATEMENT>`. Although the `<ASSIGNMENT STATEMENT>` is also a `<STATEMENT>`, it cannot be used as a label for an AWAIT statement (see Appendix A) as a SEND statement could because the grammar requires a `<BUFFER STATEMENT>` in that place.

It is the responsibility of the editor to check these subset relations before allowing the use of any saved
Gypsy segment. The computation of these subset relations is also done dynamically by the editor using the editor's grammar definition in the table. In this way, the grammar used by the editor can be defined without regard for these subset relations and the editor will correctly compute the required partial ordering.

The uses of the workspace are primarily involved in the moving or copying of program segments from one place to another. The workspace exists beyond the current editing session and can therefore be used to move or copy program parts from one routine to another. A requirement of verification that is commonly violated is that the routine bodies be small and simple. The difficulty of proving complicated verification conditions forces the factoring of the proof, and therefore the program, into small pieces for analysis. Since the Gypsy language is a structured language, it is very easy to replace a compound statement by a procedure call, thereby simplifying the routine. This can be accomplished in a straightforward way using the workspace feature of the Gypsy Structure Editor. The compound statement is saved in the workspace, then replaced in the routine with a procedure call. A new procedure is created and the saved compound statement is made the body of the procedure using the editor. An example of the use of a previously saved Gypsy segment is the following:
4.5.3 The Editor-user Interaction

Besides the display, editing, and workspace commands described above, there are a few other commands available to the user to control the editing process.

The user begins the editing process by directing the Gypsy Verification Environment to invoke the editor on a specified Gypsy unit. Editing is normally begun with the EDIT command, but the editor can also be invoked with the EXAMINE command which allows the specified unit to be examined by the user, but not modified.

The edit session is ended by directing the editor to return the unit to the database manager and to return control to the Gypsy Programming System toplevel. This normal return is done with the EXIT command. The edit session can also be terminated by the ABORT command which returns control without returning the unit to the database.
All of the modifications done are effectively lost.

A capability to undo the effects of a series of editing modifications is a major convenience to the user and can greatly increase the user's confidence in the editor since the user knows that blunders can be corrected. The ABORT command can be used to undo the effects of an editing modification, but it is very coarse in that all of the other modifications done are also lost. The edit command UNDO allows the effects of recent modifications to be reversed without losing the entire edit session. This is accomplished by the periodic saving of the state of the unit and allowing the user to back up to the last saved state by using the UNDO command. It is possible to back up to earlier and earlier saved states by repeated use of UNDO. The saving of the edit state is done each time the user performs a DOWN command. This not only allows the user to know exactly what the effects of an UNDO command would be, but also reduces the overhead of saving the state of the edit. This is because only the program segment which becomes the new focus has to be saved since that will be the only part which can be modified after the DOWN command is done.

It is sometimes convenient to pause in the editing session to print or edit another unit. All of the commands available to the user at the system toplevel are made
available to the user while in the editor by creating an inferior fork consisting of the executive toplevel. This EXEC command is very similar to the TENEX "exec" command. As with the TENEX operating system, an inferior exec is dismissed with the QUIT command. This feature, combined with the workspace aids, allows the user to move or copy program segments from one unit to another easily.

4.5.4 Composite Editing Functions

All of the editing commands presented above perform relatively simple basic functions. They form a set of commands which allow most desired modifications to be done in a straightforward way. The composite editing functions are internal combinations of these basic commands which allow the user to do common actions more simply. At the writing of this document, the composite functions described here have not been implemented, but this implementation is not considered to be difficult.

Only two basic movement commands have been described above: the UP and DOWN commands. In terms of movement through the syntax tree, these are the only commands needed to move to any node. The composite commands NEXT and BACK allow movement across the branches of a tree. The list of items in a grammar production can be examined one by one
using these commands.

Finding an item somewhere in the program is a more complicated form of the composite movement command. The FIND command first requires that the item be found, then the appropriate series of UP and DOWN commandes are performed internally. The FIND command accepts a statement type or an identifier from the user, then changes the focus from the current one to the next occurrence of the statement or identifier.

The three basic modification commands, REPLACE, DELETE, and INSERT, along with the workspace, allow most desired modifications to be done fairly easily. However, some common editing actions can be turned into new composite commands. The MOVE command moves a program segment from one place to another. This is equivalent to doing a SAVE, a DELETE, a series of attention-changing commands, then a REPLACE or INSERT command. The MOVE command looks like the REPLACE or INSERT command, except that the new program segment does not come from the parser or the workspace but from another place in the program.

Another possible composite function is the SUBSTITUTE command. The easiest form of the SUBSTITUTE command only allows substitutions of identifiers, but it is possible to allow substitutions of arbitrary constructs.
CHAPTER FIVE

EDITOR PROBLEMS AND EXTENSIONS

A few inherent problems and limitations of this type of editor have been discovered in the design and implementation of the Gypsy Structure Editor. A few extensions of the editor are proposed to minimize some of these limitations.

5.1 The Token Editor As Text Editor

One severe inherent limitation of a structure editor is its incapability of doing text editing-like changes. Some editing changes are much simpler when the full versatility of the text editor can be used. Most often these kinds of changes pertain to modifying the block structure of the language. An error in the original entry, for example, might require the addition of an "END" construct somewhere in the program. The result would be a major change in the block structure of the program and hence in its internal form, although the textual change would be minimal. This kind of modification would be most difficult using the Gypsy Structure Editor.
The editor works with the parsed internal form of the program and the Gypsy System has no internal form corresponding to a syntactically incorrect program. Therefore any program or program segment introduced to the system which contains syntax errors must be represented in a textual form and edited with a text editor or equivalent.

One extension to the editor would be to allow editing of a textual form of the program. The Gypsy language defines a set of tokens with which programs are defined. The textual form of the program which would be used is the string of tokens which make up the program. There exists a string of tokens even for a syntactically incorrect program. This string of tokens can be generated from the internal form by the pretty print package. The editing commands would be very similar to a text editor with the tokens handled as characters. Because of the kind of modifications allowed by the token editor, the syntactic correctness of a program segment is no longer guaranteed. The parser is needed to check the syntax and convert the token string into the internal form.

While editing a program unit, the token editor can be used on any program segment. The current focus is converted to a token string using the pretty print package. The token string is edited with the token editor and then, before the editor can continue, the token string must be
sent through the parser. The internal form returned by the parser replaces the original internal form. If the parser fails to successfully translate the program segment, then the token string must be further edited, or the process aborted.

The token editor can be used independently of the structure editor whenever the parser is used. When the parser fails to parse some program text, the token editor can be invoked to edit the token string and correct the text. This can even be done when the parser accepts input from the user while executing a command in the Gypsy Structure Editor.

5.2 Failures Of The Grammar Definition

Many languages, and in particular the Gypsy language, have context sensitive aspects of the language although the grammar form of the language is context free. The context sensitive parts of the Gypsy language have been put into a LR(1) grammar by enumerating all of the possible combinations of the context sensitive parts.

An example of this in the Gypsy grammar is the constant definition:

\[
\text{CONST } <\text{IDENTIFIER}> : <\text{TYPE REF}> = <\text{EXPRESSION}>
\]
CONST <IDENTIFIER> : <TYPE REF> = PENDING
CONST <IDENTIFIER> = <EXPRESSION>

The type reference of the constant definition is allowed to be omitted if the value of the constant is present (not pending).

In these cases, the user correctly views this as one production:

CONST <IDENTIFIER> : <CONST TYPE> = <CONST VALUE>

where:

<CONST TYPE> ::= <TYPE REF>
   EMPTY
<CONST VALUE> ::= <EXPRESSION>
   PENDING

with the condition that the <CONST TYPE> cannot be empty unless the <CONST VALUE> is not PENDING.

Since the actual grammar definition does not reflect the user's view of the language, the editor cannot normally allow modifications which seem natural to the user. In the above example, the user should be able to modify a constant definition by deleting the type reference part.

The current Gypsy Structure Editor allows such modifications by using special entries in the editor table. As the grammar definition is put into the editor table, such
productions are changed to the more compact context-sensitive form and special functional entries are put in the table to check the context sensitive parts. The functional entries are put under the INSERT and DELETE tests described above.

This solution to the problem succeeds in spite of the grammar definition, not because of it. An extension to the editor would be to modify the table and algorithm to handle a context-sensitive grammar which would allow natural modifications of programs.

5.3 The Powerful Uses Of A Structure Editor

The design and implementation of the Gypsy Structure Editor has been described above. This section is an overview and summary of the many benefits of a structured syntax editor in a complete programming environment.

5.3.1 Creation of a Program Unit

A Gypsy routine definition, like the routine definitions of many other block structured algorithmic languages, has a complex syntactic definition. A Gypsy routine has a routine header, routine specifications, local declarations, and the routine body. The various structured
statements have fairly complex forms whose exact syntax are often forgotten by the user.

The editor can be used to create a program unit and, since the editor knows the structure of all program constructs, it can be used to help the user in writing the program. The editor displays a Gypsy segment as a set of terminal and non-terminal symbols, and those Gypsy segments corresponding to the non-terminals that are missing are represented as the non-terminals in comments. In this way, a template of a routine or compound program statement can be presented to the user with only the non-terminal parts to be filled in. It is not necessary for the user to remember the particular order of the elements, or to remember where the commas, semi-colons, and periods go.

The editor can present the creation of a program unit as a menu selection process where the order of how things are filled in is arbitrary. This allows the user to easily return to the local declaration section to declare a new local variable that was just discovered to be needed. In fact, it is possible to add commands to the editor to facilitate such common uses as adding new routine parameters or local declarations without having to return to that part of the program.
The major benefits of this method of creating a program unit are its easy use and immediate feedback. It is not necessary for the user to constantly reference the language manual since the editor presents the form of the program to the user. The user receives immediate feedback regarding syntactic correctness of the program, thereby drastically shortening the annoying cycle time of program writing and syntax error detection. In addition, the semantic correctness of the program can easily be checked by leaving the editor and calling up the parser component. The editor can then be reentered for further modification.

5.3.2 Ease of Modification

In a highly structured language like Gypsy, the main problem of editing is modifying and maintaining the structure of the program. Since the Gypsy Structure Editor understands the structure of the language, this task is greatly simplified. The readability of a program written in a highly structured language is dependent to a large extent on the form of the program, its indentations and spacing. Using a normal text editor to extensively modify the original program can destroy the carefully constructed form of the program, thereby reducing its readability and, hence, its understandability. Since the internal form of the program being edited is always printed anew with the pretty
print package, the external form of the program is always well-structured and readable.

One of the first basic tasks in writing and editing a program is to get the syntax correct. Syntax errors usually do not reflect logical errors in the program or a lack of understanding of the language, but usually are the result of a lack of familiarity with certain syntactic constructs or sloppiness. The long delay between editing, syntactic checking, and editing again is time consuming and irritating. A syntax preserving editor gives immediate feedback as to syntax correctness and reduces this delay. The Gypsy Structure Editor also presents templates of the syntactic constructs to the user, automatically filling in the terminal symbols required. In this way, it is possible for a user to edit a Gypsy program with only a minimal knowledge of the syntax.

Because the editor only need check the syntax of the actual modifications, it is not necessary to re-check the syntax of the entire program or even the program unit. If this process of syntax checking is time consuming or expensive in system resources, then the minimal work done by the editor can be a distinct advantage.

The Gypsy Structure Editor provides commands which are not usually found in normal text editors. The main
advantage is the capability of directing editing commands toward syntactic constructs rather than characters or lines. The two most unusual and useful commands are the UNDO and WORKSPACE commands. Some editors have the capability of periodically saving the state of the edit, but that usually involves backing up the entire file, which is considerably more expensive than the method used by this editor. Some editors have "registers" in which text can be saved, but the workspace is a far more versatile in that it saves Gypsy segments and can save an unbounded number of segments under user-defined names.

Since the Gypsy Structure Editor is a part of its programming environment, the Gypsy Verification Environment, it can draw upon all of the powers of the entire environment. The most important uses of this environment are the syntax checking done whenever the user inputs a program segment and the pretty printing done whenever the user requests a display of the current program focus. Other benefits are the capability of displaying program units other than the one being edited and the ability to have the unit semantically checked by the parser.
5.3.3 Incremental Development

As more and more analysis is performed on a program, the more expensive becomes any changes to that program. Incremental development is the process of minimizing the effects of a modification to some part of the program. The more knowledge that the incremental development module has about the actual changes made, the better it can do its job of minimizing effects.

If the editing of the program is done outside of the programming environment which includes the incremental development component, then the changes made must be inferred by the system. This process is either incomplete or very expensive to perform. The system editor allows the incremental development process to know exactly what changes were made. This information is compiled easily and cheaply by the editor, thereby greatly decreasing the overhead of incremental development.

5.3.4 The Editor and Program Modification

The editor can not only document modifications for various uses (as described in the previous section), but can also provide some control over the kinds of edits done, what is edited, and by whom.
In the process of recording each of the modifications done to each program unit, the editor not only aids in incremental development but can even leave a complete trail of edits which describe the past state of the system at any time and the modifications performed to transform each state of the system. The information saved can include the program unit being modified, the current date and time, the user making the modifications, and the exact changes made.

In addition to recording the history of program modifications, the editor can control these modifications. The most basic kind of control would be whether to allow certain users to edit certain program units. This could be used to prevent unauthorized modification of the database by users who are not supposed to be modifying anything. Some program units could be considered so critical that any modification to them could only be done with permission from the highest level. In terms of incremental development, it is possible to modify a critical type definition in such a way as to cause every other unit in the database to be re-checked (at a considerable expense). Such units might be protected from casual editing.

Even when a particular user is authorized to edit a unit, there might still be reasons to prevent any editing. If one user is in the process of making complicated
modifications to a program unit over a period of several days, it would be reasonable to want to restrict edits by other users during this time frame. The editor could allow such a user to put a "lock" on the unit to prevent anyone else from editing the unit during this critical period.

After an edit has been completed, the editor may either suggest or demand that certain operations regarding that unit be carried out. These operations could include making sure that there are no semantic errors by running the unit through the parser and not allowing the editing process to terminate successfully until the semantic errors are fixed. The incremental development required to bring the system to an internally consistent state following the modifications could also be suggested or mandated and perhaps documentation should be updated in the wake of an editing session. To attempt to guarantee correctness of the resulting program after editing, the editor could direct analysis or testing after the edit session.
APPENDIX A

THE GYPSY GRAMMAR DEFINITION

<PROGRAM> ::= <PROGRAM DESCRIPTION>
   / <PROGRAM DESCRIPTION> ;

<ABSTRACT EXT SPECS> ::= <ENTRY SPECS> <BLOCK SPECS>
   <EXIT SPECS>

<ABSTRACT TYPE BODY> ::= BEGIN IDENTIFIER : <TYPE DEF> ;
   <HOLD SPEC> END

<ACCESS LIST> ::= <NAMELIST> >
   / EMPTY

<ACTUAL CONDS> ::= UNLESS ( <NAMELIST> )

<ACTUAL PARAMS> ::= ( <SELECT EXP LIST> )

<ALTERATION> ::= <EACH CLAUSE> <COMPONENT ASSIGN>
   / <EACH CLAUSE> <COMPONENT DELETE>

<ALTERATION CLAUSE> ::= WITH ( <ALTERATIONS> )

<ALTERATIONS> ::= <ALTERATION>
   / <ALTERATIONS> ; <ALTERATION>

<ARRAY TYPE> ::= ARRAY ( <TYPE REF> ) OF <TYPE DEF>

<ASSERT SPEC> ::= ASSERT <SPEC EXP>

<ASSIGNMENT STMT> ::= <PLACE REF> := <EXPRESSION>

<AWAIT CASELIST> ::= <AWAIT CASE>
   / <AWAIT CASELIST> <AWAIT CASE>

<AWAIT CASE> ::= <EACH CLAUSE> ON <BUFFER STATEMENT> THEN
   <STATEMENT LIST>
<AWAIT STMT> ::= AWAIT <AWAIT CASELIST> <END CLAUSE>

<BEGIN STMT> ::= BEGIN <STATEMENT LIST> <END CLAUSE>

<BINARY OP> ::= *
  | /  
  | MOD  
  | EQ  
  | NE  
  | GE  
  | LE  
  | LT  
  | GT  
  | SUB  
  | UNION  
  | DIFFERENCE  
  | INTERSECT  
  | @  
  | IFF  
  | ->  
  | **  
  | AND  
  | OR  
  | DIV  
  | <:  
  | :>  
  | OMIT  
  | ADJOIN  
  | IN  
  | APPEND  
  | IMP  
  | &  
  | +  
  | -  
  | <  
  | >  
  | =

<Block Definitions> ::= <Dynamic Ext Specs>
  <Local Decl Part> <Keep Specs>

<Block Specs> ::= BLOCK <Nonvalidated Exp> ;
  / EMPTY

<Buffer Restrict> ::= <Input>
  / <Output>
  / EMPTY

<Buffer Statement> ::= <Send Stmt>
  / <Give Stmt>
  / <Receive Stmt>
<BUFFER TYPE> ::= BUFFER <OPT EXP> OF <TYPE DEF>

<CASE ELSE> ::= ELSE : <STATEMENT LIST>
    / EMPTY

<CASE PART> ::= IS <CASE SELECTLIST> : <STATEMENT LIST>

<CASE SELECT> ::= CHAR
    / NUMBER
    / - NUMBER
    / IDENTIFIER

<CASE SELECTLIST> ::= <CASE SELECT>
    / <CASE SELECTLIST> , <CASE SELECT>

<CASE STMT> ::= CASE <EXPRESSION> <CASEPART LIST>
    <CASE ELSE> <END CLAUSE>

<CASEPART LIST> ::= <CASE PART>
    / <CASEPART LIST> <CASE PART>

<CBLOCK SPECS> ::= CBLOCK <NONVALIDATED EXP> ;
    / EMPTY

<CENTRY SPECS> ::= CENTRY <SPEC EXP> ;
    / EMPTY

<CEXIT SPECS> ::= CEXIT <SPEC EXP> ;
    / CEXIT <EXIT CASE> ;
    / EMPTY

<COBEGIN STMT> ::= COBEGIN <COBEGIN STMTLIST> <END CLAUSE>
    / COBEGIN <COBEGIN STMTLIST> ;
    <END CLAUSE>

<COBEGIN STMTLIST> ::= <EACHCALL STMT>
    / <COBEGIN STMTLIST> ;
    <EACHCALL STMT>

<COMPONENT ASSIGN> ::= <SELECTOR LIST> ::= <EXPRESSION>
    / BEFORE <SELECTOR LIST> ::= <EXPRESSION>
    / BEHIND <SELECTOR LIST> ::= <EXPRESSION>

<COMPONENT DELETE> ::= SEQOMIT <SELECTOR LIST>
    / MAPOMIT <SELECTOR LIST>

<CONCRETE EXT SPECS> ::= <CENTRY SPECS> <CBLOCK SPECS>
    <CEXIT SPECS>
<CONCURRENT CONTROL STMT> ::= <COBEGIN STMT> / <AWAIT STMT>

<COND DECL> ::= COND <NAMELIST>

<CONDITION CLAUSE> ::= IS <NAMELIST> : <NONVALIDATED EXP> / IS NORMAL : <SPEC EXP>

<CONDITION LIST> ::= <CONDITION CLAUSE> / <CONDITION LIST> ; <CONDITION CLAUSE>

<CONST DECL> ::= CONST IDENTIFIER : <TYPE REF> = <EXPRESSION> / CONST IDENTIFIER : <TYPE REF> = PENDING / CONST IDENTIFIER = <EXPRESSION>

<CONSTRUCTOR> ::= ( <SELECT EXP> , <SELECT EXP LIST> ) / ( <EXPRESSION> .. <EXPRESSION> ) / ( SET : <SELECT EXP LIST> ) / ( SEQ : <SELECT EXP LIST> )

<DYNAMIC COMP REF> ::= <EXPRESSION> / <EXPRESSION> FROM <VARIABLE REF>

<DYNAMIC EXT SPECS> ::= <ABSTRACT EXT SPECS> <CONCRETE EXT SPECS>

<EACH CLAUSE> ::= EACH IDENTIFIER : <RANGE> , / EMPTY

<EACHCALL STMT> ::= <EACH CLAUSE> <ROUTINE CALL>

<ELIF LIST> ::= <ELIF PART> / <ELIF LIST> <ELIF PART>

<ELIF PART> ::= ELIF <EXPRESSION> THEN <STATEMENT LIST>

<ELSE PART> ::= ELSE <STATEMENT LIST> / EMPTY

<END CLAUSE> ::= END / <WHEN PART> END

<ENTRY SPECS> ::= ENTRY <SPEC EXP> ; / EMPTY

<EXIT CASE> ::= CASE ( <CONDITION LIST> )

<EXIT SPECS> ::= EXIT <SPEC EXP> ; / EXIT <EXIT CASE> ;
/ EMPTY

<EXPRESSION> ::= <EXP1>
   / <EXP2>

<EXP1> ::= <VALUE REF>
   / <UNARY OP> <VALUE REF>
   / <QUANTIFIED EXP>
   / <UNARY OP> <QUANTIFIED EXP>

<EXP2> ::= <VALUE REF> <BINARY OP> <EXP3>
   / <UNARY OP> <VALUE REF> <BINARY OP> <EXP3>

<EXP3> ::= <EXP1>
   / <EXP2>

<EXTENDING> ::= IDENTIFIER

<EXTENDS> ::= EXTENDS STRING
   / EMPTY

<FORMAL COND PARAMS> ::= UNLESS ( COND <NAMELIST> )
   / UNLESS ( <NAMELIST> )
   / EMPTY

<FUNCTION> ::= FUNCTION IDENTIFIER <EXTENDS>

<FUNCTION DATA PARAMS> ::= ( <FUNCTION PARAMLIST> )

<FUNCTION HEADER> ::= <FUNCTION> <FUNCTION DATA PARAMS> :
   <TYPE REF> <FORMAL COND PARAMS>
   / <FUNCTION> : <TYPE REF>
   <FORMAL COND PARAMS>

<FUNCTION PARAM> ::= CONST <NAMELIST> : <TYPE REF>
   / <NAMELIST> : <TYPE REF>

<FUNCTION PARAMLIST> ::= <FUNCTION PARAM>
   / <FUNCTION PARAMLIST> ;
   <FUNCTION PARAM>

<GIVE STMT> ::= GIVE <DYNAMIC COMP REF> TO <VARIABLE REF>

<HOLD SPEC> ::= HOLD <NONVALIDATED EXP>
   / HOLD <NONVALIDATED EXP> ;

<IF EXP> ::= IF <EXPRESSION> THEN <EXPRESSION> ELSE
   <EXPRESSION> FI

<IF STMT> ::= IF <THEN PART> <ELSE PART> <END CLAUSE>
   / IF <THEN PART> <ELIF LIST> <ELSE PART>
<END CLAUSE>

<INITIAL SPEC> ::= INITIALLY ( <EXPRESSION> )
    / INITIALLY ( PROVE <EXPRESSION> )
    / INITIALLY ( ASSUME <EXPRESSION> )
    / EMPTY

<INITIALIZATION> ::= := <EXPRESSION>
    / := PENDING
    / EMPTY

<KEEP SPECS> ::= KEEP <SPEC EXP> ;
    / EMPTY

<LEAVE STMT> ::= LEAVE

<LEMMA DECL> ::= <LEMMA HEADER> = <NONVALIDATED EXP>

<LEMMA HEADER> ::= LEMMA IDENTIFIER <FUNCTION DATA PARAMS>
    / LEMMA IDENTIFIER

<LITERAL CONST> ::= CHAR
    / STRING
    / NUMBER

<LOCAL DECL> ::= <VAR DECL> ;
    / <CONST DECL> ;
    / <COND DECL> ;

<LOCAL DECL LIST> ::= <LOCAL DECL>
    / <LOCAL DECL LIST> <LOCAL DECL>

<LOCAL DECL PART> ::= <LOCAL DECL LIST>
    / EMPTY

<LOOP STMT> ::= LOOP <STATEMENT LIST> <END CLAUSE>

<MAPPING TYPE> ::= MAPPING <OPT EXP> FROM <TYPE DEF> TO
    <TYPE DEF>

<MOVE STMT> ::= MOVE <DYNAMIC COMP REF> TO <PLACE REF>
    / MOVE <DYNAMIC COMP REF> INTO
    <VARIABLE REF>

<NAME DECL> ::= NAME <RENAME LIST> FROM IDENTIFIER

<NAMELIST> ::= IDENTIFIER
    / <NAMELIST>, IDENTIFIER

<NONVALIDATED EXP> ::= <EXPRESSION>
    / ( <NONVALIDATED EXP ITEMLIST> )
<NONVALIDATED EXP ITEM> ::= PROVE <EXPRESSION>
   / ASSUME <EXPRESSION>

<NONVALIDATED EXP ITEMLIST> ::= <NONVALIDATED EXP ITEM>
   / <NONVALIDATED EXP ITEMLIST> ;
   <NONVALIDATED EXP ITEM>

<OPT EXP> ::= ( <EXPRESSION> )
   / EMPTY

<PARAM KIND> ::= CONST
   / VAR
   / EMPTY

<PLACE REF> ::= <VARIABLE REF>
   / BEFORE <SEQ COMP REF>
   / BEHIND <SEQ COMP REF>

<PROCEDURAL STMT> ::= <ROUTINE CALL>
   / <ASSIGNMENT STMT>
   / <REMOVE STMT>
   / <MOVE STMT>
   / <SEND STMT>
   / <GIVE STMT>
   / <RECEIVE STMT>

<PROCEDURE DATA PARAMS> ::= ( <PROCEDURE PARAMLIST> )

<PROCEDURE HEADER> ::= PROCEDURE IDENTIFIER
   <PROCEDURE DATA PARAMS>
   <FORMAL COND PARAMS>

<PROCEDURE PARAMLIST> ::= <PROCEDURE PARAM>
   / <PROCEDURE PARAMLIST> ;
   <PROCEDURE PARAM>

<PROCEDURE PARAM> ::= <PARAM KIND> <NAMELIST> : <TYPE REF>

<PROGRAM DESCRIPTION> ::= <SCOPE EXT OR DECL>
   / <PROGRAM DESCRIPTION> ;
   <SCOPE EXT OR DECL>

<QUANTIFIED EXP> ::= ALL <NAMELIST> : <TYPE REF> ,
   <EXPRESSION>
   / SOME <NAMELIST> : <TYPE REF> ,
   <EXPRESSION>

<RANGE> ::= ( <EXPRESSION> .. <EXPRESSION> )

<RECEIVE STMT> ::= RECEIVE <VARIABLE REF> FROM
<VARIABLE REF>

<RECORD COMP> ::= <NAMELIST> : <TYPE DEF>

<RECORD COMP LIST> ::= <RECORD COMP>
                / <RECORD COMP LIST> ; <RECORD COMP>

<RECORD TYPE> ::= RECORD ( <RECORD COMP LIST> )

<REMOVE STMT> ::= REMOVE <DYNAMIC COMP REF>

<RENAME> ::= IDENTIFIER = IDENTIFIER
       / IDENTIFIER

<RENAME LIST> ::= <RENAME>
                / <RENAME LIST> , <RENAME>

<Routine BODY> ::= BEGIN <BLOCK DEFINITIONS>
                <STATEMENT LIST> <END CLAUSE>

<Routine CALL> ::= IDENTIFIER <ACTUAL PARAMS>
                / IDENTIFIER <ACTUAL PARAMS>
                <ACTUAL CONDS>

<Routine DECL> ::= <Routine HEADER> = <Routine BODY>
                / <Routine HEADER> = PENDING

<Routine HEADER> ::= <PROCEDURE HEADER>
                / <FUNCTION HEADER>

<SCALAR TYPE> ::= ( <NAMELIST> )

<SCOPE BODY> ::= BEGIN <UNIT-NAME DECL LIST> END
                / BEGIN <UNIT-NAME DECL LIST> ; END

<SCOPE DECL> ::= <SCOPE HEADER> = <SCOPE BODY>

<SCOPE EXT OR DECL> ::= <ACCESS LIST> <SCOPE DECL>
                     / COMMAND <EXTENDING> <ACCESS LIST>
                     <SCOPE DECL>

<SCOPE HEADER> ::= SCOPE IDENTIFIER

<Select EXP> ::= <EXPRESSION>
       / <EXPRESSION> .. <EXPRESSION>

<Select EXP LIST> ::= <Select EXP>
                / <Select EXP LIST> , <Select EXP>

<SELECTOR LIST> ::= <SELECTOR>
                / <SELECTOR LIST> <SELECTOR>
<SELECTOR> ::= . IDENTIFIER
             / ( <SELECT EXP LIST> )

<SELECT-ALTER LIST> ::= <SELECT-ALTER>
                        / <SELECT-ALTER LIST> <SELECT-ALTER>

<SELECT-ALTER> ::= <SELECTOR>
                   / <ALTERATION CLAUSE>

<SEND STMT> ::= SEND <EXPRESSION> TO <VARIABLE REF>

<SEQ COMP REF> ::= IDENTIFIER <SELECTOR LIST>

<SEQ TYPE> ::= SEQUENCE <OPT EXP> OF <TYPE DEF>

<SEQUENTIAL CONTROL STMT> ::= <IF STMT>
                              / <CASE STMT>
                              / <LOOP STMT>
                              / <BEGIN STMT>
                              / <LEAVE STMT>
                              / <SIGNAL STMT>

<SET TYPE> ::= SET <OPT EXP> OF <TYPE DEF>

<SIGNAL STMT> ::= SIGNAL IDENTIFIER

<SPEC EXP> ::= <EXPRESSION>
              / <EXPRESSION> OTHERWISE IDENTIFIER
              / ( <SPEC EXP ITEMLIST> )

<SPEC EXP ITEM> ::= PROVE <EXPRESSION>
                  / ASSUME <EXPRESSION>
                  / PROVE <EXPRESSION> OTHERWISE IDENTIFIER
                  / ASSUME <EXPRESSION> OTHERWISE IDENTIFIER

<SPEC EXP ITEMLIST> ::= <SPEC EXP ITEM>
                        / <SPEC EXP ITEMLIST> ;
                        / <SPEC EXP ITEM>

<STATEMENT> ::= <PROCEDURAL STMT>
               / <SEQUENTIAL CONTROL STMT>
               / <CONCURRENT CONTROL STMT>

<STATEMENT LIST> ::= <STMT ASSERT LIST>
                    / <STMT ASSERT LIST> ;
                    / PENDING
                    / PENDING ;
                    / ;
                    / EMPTY
<STMT ASSERT> ::= <STATEMENT>  
   / <ASSERT SPEC>

<STMT ASSERT LIST> ::= <STMT ASSERT>  
   / <STMT ASSERT LIST> ; <STMT ASSERT>

<STRUCTURED TYPE> ::= <SET TYPE>  
   / <MAPPING TYPE>  
   / <SEQ TYPE>  
   / <ARRAY TYPE>  
   / <RECORD TYPE>

<THEN PART> ::= <EXPRESSION> THEN <STATEMENT LIST>

<TYPE BODY> ::= <TYPE DEF>  
   / <ABSTRACT TYPE BODY>  
   / PENDING

<TYPE DECL> ::= <TYPE HEADER> = <TYPE BODY>

<TYPE DEF> ::= <TYPE REF>  
   / <TYPE REF> ::= <EXPRESSION>  
   / <SCALAR TYPE>  
   / <STRUCTURED TYPE>  
   / <BUFFER TYPE>

<TYPE HEADER> ::= TYPE IDENTIFIER <INITIAL SPEC>  
   / <ACCESS LIST>

<TYPE REF> ::= IDENTIFIER <BUFFER RESTRICT>  
   / <RANGE>  
   / IDENTIFIER <RANGE>

<UNARY OP> ::= -  
   / NOT

<UNIT DECL> ::= <ROUTINE DECL>  
   / <LEMMAT DECL>  
   / <TYPE DECL>  
   / <CONST DECL>

<UNIT-NAME DECL> ::= <ACCESS LIST> <UNIT DECL>  
   / <NAME DECL>

<UNIT-NAME DECL LIST> ::= <UNIT-NAME DECL>  
   / <UNIT-NAME DECL LIST> ; <UNIT-NAME DECL>

<VALUE REF> ::= <LITERAL CONST>  
   / ( <EXPRESSION> )
/ ( <EXPRESSION> ) <SELECT-ALTER LIST>
/ <IF EXP>
/ <IF EXP> <SELECT-ALTER LIST>
/ IDENTIFIER
/ IDENTIFIER <ACTUAL CONDS>
/ IDENTIFIER <ACTUAL CONDS>
/ <SELECT-ALTER LIST>
/ IDENTIFIER <SELECT-ALTER LIST>
/ IDENTIFIER <SELECT-ALTER LIST>
/ <ACTUAL CONDS>
/ IDENTIFIER <SELECT-ALTER LIST>
/ <ACTUAL CONDS> <SELECT-ALTER LIST>
/ <CONSTRUCTOR>
/ <CONSTRUCTOR> <SELECT-ALTER LIST>

<VAR DECL> ::= VAR <NAMELIST> : <TYPE REF>
/ <INITIALIZATION>
/ VAR <NAMELIST> := <EXPRESSION>

<VARIABLE REF> ::= IDENTIFIER <SECTOR LIST>
/ IDENTIFIER

<WHEN CASE> ::= IS <NAMELIST> : <STATEMENT LIST>

<WHEN CASELIST> ::= <WHEN CASE>
/ <WHEN CASELIST> <WHEN CASE>

<WHEN ELSE> ::= ELSE : <STATEMENT LIST>
/ EMPTY

<WHEN PART> ::= WHEN <WHEN CASELIST> <WHEN ELSE>
/ WHEN <WHEN ELSE>
APPENDIX B

THE EDITOR GRAMMAR DEFINITION

<ABSTRACT EXTERNAL SPECIFICATIONS> ::=  
  <ENTRY SPECIFICATION>  
  <BLOCK SPECIFICATION>  
  <EXIT SPECIFICATION>

<ACTUAL COND PARAMETERS> ::=  Unless ( <IDENTIFIER>* )

<ACTUAL DATA PARAMETERS> ::=  ( <EXPRESSION>* )

<ALTER PLACE REF> ::=  <BEFORE/BEHIND> <SELECTOR LIST>

<ALTERATION CLAUSE> ::=  With ( <ALTERATION PART>* )

<ALTERATION PART> ::=  <EACH CLAUSE> <COMPONENT ALTERATION>

<ASSERT SPECIFICATION> ::=  Assert ( <SPECIFICATION PART>* )

<ASSIGNMENT STATEMENT> ::=  <PLACE REF> := <EXPRESSION>

<AWAIT CASE> ::=  <EACH CLAUSE> On <BUFFER STATEMENT> then  
  <STATEMENT LIST>

<AWAIT LIST> ::=  <AWAIT CASE>*

<AWAIT STATEMENT> ::=  Await <AWAIT LIST> <END CLAUSE>

<BEGIN STATEMENT> ::=  Begin <STATEMENT LIST> <END CLAUSE>

<BLOCK SPECIFICATION> ::=  Block ( <NONVALIDATED PART>* )

<BUFFER STATEMENT> ::=  <SEND STATEMENT> | 
  <GIVE STATEMENT> | 
  <RECEIVE STATEMENT>

<CASE> ::=  is <CASE LABEL LIST> : <STATEMENT LIST>
<CASE ELSE> ::= Else : <STATEMENT LIST>

<CASE LABEL LIST> ::= <CASE LABEL>*

<CASE LIST> ::= <CASE>*

<CASE STATEMENT> ::= Case <EXPRESSION> <CASE LIST>
                    <CASE ELSE> <END CLAUSE>

<CBLOCK SPECIFICATION> ::= Cblock ( <NONVALIDATED PART>* )

<CENTRY SPECIFICATION> ::= Centry ( <SPECIFICATION PART>* )

<CEXIT SPECIFICATION> ::= Cexit case ( <EXIT CASE ELEMENT>* )
                          Cexit ( <SPECIFICATION PART>* )

<COBEGIN STATEMENT> ::= Cobegin <EACHCALL STATEMENTLIST>
                        <END CLAUSE>

<COMPONENT ALTERATION> ::=<ALTER PLACE REF> ::= <EXPRESSION>
                          <KIND OMIT> <SELECTOR LIST>

<CONCRETE EXTERNAL SPECIFICATIONS> ::=<CENTRY SPECIFICATION>
                                      <CBLOCK SPECIFICATION>
                                      <CEXIT SPECIFICATION>

<CONST DECLARATION> ::= Const <IDENTIFIER> : <TYPE REF> =
                        <CONST EXP>

<CONST EXP> ::= Pending |
               <EXPRESSION>

<CONST UNIT> ::= <UNIT ACCESS LIST> <CONST DECLARATION>

<CONSTRUCTOR EXPRESSION> ::= <EXPRESSION> .. <EXPRESSION>
                            <EXPRESSION>

<CONSTRUCTOR EXPRESSIONS> ::= <CONSTRUCTOR EXPRESSION>*

<DYNAMIC COMPONENT REF> ::= <VARIABLE REF>
                          <EXPRESSION> From <VARIABLE REF>

<EACH CLAUSE> ::= Each <IDENTIFIER> : <RANGE>,

<EACHCALL> ::= <PROCEDURE CALL STATEMENT> |
               <EACH CLAUSE> <PROCEDURE CALL STATEMENT>

<EACHCALL STATEMENTLIST> ::= <EACHCALL>*
<ELIF PART> ::= Elif <EXPRESSION> then <STATEMENT LIST>

<ELIF PART LIST> ::= <ELIF PART>*

<END CLAUSE> ::= When <WHEN CASE LIST> <WHEN ELSE> End

<ENTRY SPECIFICATION> ::= Entry (<SPECIFICATION PART>* )

<EXIT CASE ELEMENT> ::= Is Normal : <SPECIFICATION EXPRESSION> | Is <NAMELIST> : <NONVALIDATED EXP>

<EXIT SPECIFICATION> ::= Exit case (<EXIT CASE ELEMENT>* ) | Exit (<SPECIFICATION PART>* )

<EXPRESSION> ::= <EXPRESSION> <BINARY OP> <EXPRESSION> | <UNARY OP> <EXPRESSION> | <QUANTIFIED KIND> <IDENTIFIER> : <TYPE REF> , <EXPRESSION> | If <EXPRESSION> then <EXPRESSION> else <EXPRESSION> Fi | ( <CONSTRUCTOR KIND> : <CONSTRUCTOR EXPRESSIONS> ) | <EXPRESSION> <SELECT-ALTER-COND LIST> | <SIMPLE VALUE>

<EXTENDS> ::= Extends <STRING>

<FORMAL COND PARAMS> ::= Unless (<IDENTIFIER>* )

<FUNCTION DATA PARAMETER> ::= Const <IDENTIFIER> : <TYPE REF>

<FUNCTION DATA PARAMETERS> ::= ( (<FUNCTION DATA PARAMETER>* )

<FUNCTION DECLARATION> ::= <FUNCTION HEADER> = <ROUTINE BODY>

<FUNCTION HEADER> ::= <UNIT ACCESS LIST> Function <IDENTIFIER> <EXTENDS> <FUNCTION DATA PARAMETERS> : <TYPE REF> <FUNCTION DATA PARAMETERS>

<GIVE STATEMENT> ::= Give <DYNAMIC COMPONENT REF> To <VARIABLE REF>

<IF STATEMENT> ::= If <EXPRESSION> then <STATEMENT LIST> <ELIF PART LIST> Else
\texttt{<STATEMENT LIST> <END CLAUSE>}

\texttt{<INITIAL SPECIFICATION> ::= <PROVE/ASSUME> <EXPRESSION>}

\texttt{<KEEP SPECIFICATION> ::= \textit{Keep} ( <SPECIFICATION PART>*)}

\texttt{<LEMMA DECLARATION> ::= <LEMMA HEADER> = <NONVALIDATED EXP>}

\texttt{<LEMMA HEADER> ::= <UNIT ACCESS LIST> Lemma <IDENTIFIER>}

\texttt{<FUNCTION DATA PARAMETERS>}

\texttt{<LOCAL DECLARATION> ::= \texttt{Var} <IDENTIFIER> : <TYPE REF> ::=}

\texttt{<EXPRESSION> |}

\texttt{<CONST DECLARATION> |}

\texttt{Cond <IDENTIFIER>}

\texttt{<LOCAL DECLARATIONS> ::= <LOCAL DECLARATION>*}

\texttt{<LOOP STATEMENT> ::= Loop <STATEMENT LIST> <END CLAUSE>}

\texttt{<MOVE DESTINATION> ::= To <PLACE REF> |}

\texttt{Into <VARIABLE REF>}

\texttt{<MOVE STATEMENT> ::= Move <DYNAMIC COMPONENT REF>}

\texttt{<MOVE DESTINATION>}

\texttt{<NAMELIST> ::= <IDENTIFIER>*}

\texttt{<NONVALIDATED EXP> ::= ( <NONVALIDATED PART>*)}

\texttt{<NONVALIDATED PART> ::= <PROVE/ASSUME> <EXPRESSION>}

\texttt{<PLACE REF> ::= <BEFORE/BEHIND> <IDENTIFIER> <SELECTOR LIST>}

\texttt{<PROCEDURE CALL STATEMENT> ::= <IDENTIFIER>}

\texttt{<ACTUAL DATA PARAMETERS>}

\texttt{<ACTUAL COND PARAMETERS>}

\texttt{<PROCEDURE DATA PARAM> ::=}

\texttt{<PARAMKIND> <IDENTIFIER> : <TYPE REF>}

\texttt{<PROCEDURE DATA PARAMETERS> ::= ( <PROCEDURE DATA PARAM>* )}

\texttt{<PROCEDURE DECLARATION> ::=}

\texttt{<PROCEDURE HEADER> = <ROUTINE BODY>}

\texttt{<PROCEDURE HEADER> ::=}

\texttt{<UNIT ACCESS LIST> Procedure <IDENTIFIER>}

\texttt{<PROCEDURE DATA PARAMETERS>}

\texttt{<FORMAL COND PARAMS>
<RANGE> ::= [ <EXPRESSION> .. <EXPRESSION> ]

<RECEIVE STATEMENT> ::= Receive <VARIABLE REF>
        from <VARIABLE REF>

<RECORD ENTRY> ::= <IDENTIFIER> : <TYPE DEFINITION>

<REMOVE STATEMENT> ::= Remove <DYNAMIC COMPONENT REF>

<Routine BODY> ::= Begin <ABSTRACT EXTERNAL SPECIFICATIONS>
        <CONCRETE EXTERNAL SPECIFICATIONS>
        <LOCAL DECLARATIONS> <KEEP SPECIFICATION>
        <STATEMENT LIST> <END CLAUSE>

<Select-alter-cond> ::= <SELECTOR CLAUSE> |
        <ALTERATION CLAUSE> |
        Unless <NAMELIST>

<Select-alter-cond list> ::= <SELECT-ALTER-COND>*

>Selector clause> ::= . <IDENTIFIER> |
        ( <CONSTRUCTOR EXPRESSIONS> )

>Selector list> ::= <SELECTOR CLAUSE>*

<Send statement> ::= Send <EXPRESSION> To <VARIABLE REF>

<Signal statement> ::= Signal <IDENTIFIER>

<Specification expression> ::= ( <SPECIFICATION PART>* )

<Specification part> ::= <PROVE/ASSUME> <EXPRESSION>
        Otherwise <IDENTIFIER>

<Statement> ::= <PROCEDURE CALL STATEMENT> |
        <ASSIGNMENT STATEMENT> |
        <REMOVE STATEMENT> |
        <MOVE STATEMENT> |
        <BUFFER STATEMENT> |
        <IF STATEMENT> |
        <CASE STATEMENT> |
        <LOOP STATEMENT> |
        <BEGIN STATEMENT> |
        Leave |
        <SIGNAL STATEMENT> |
        <COBEGIN STATEMENT> |
        <AWAIT STATEMENT> |
        <ASSERT SPECIFICATION>

<Statement list> ::= Pending |
<STATEMENT>*

<TYPE BODY> ::= Pending |
             Begin <IDENTIFIER> : <TYPE DEFINITION> ;
             Hold <NONVALIDATED EXP> End |
             <TYPE DEFINITION>

<TYPE DECLARATION> ::= <TYPE HEADER> = <TYPE BODY>

<TYPE DEFINITION> ::= <TYPE HEADER> |
                    ( <NAMELIST> ) |
                    Record ( <RECORD ENTRY> ) |
                    Array ( ARRAYTYPREFERENCE ) of <TYPE DEFINITION> |
                    Set <TYPE SIZE> of <TYPE DEFINITION> |
                    Sequence <TYPE SIZE> of <TYPE DEFINITION> |
                    Mapping <TYPE SIZE> from <TYPE DEFINITION> |
                    to <TYPE DEFINITION> |
                    Buffer <TYPE SIZE> of <TYPE DEFINITION>

<TYPE HEADER> ::= <UNIT ACCESS LIST> Type <IDENTIFIER>
                Initially ( <INITIAL SPECIFICATION> )
                <UNIT ACCESS LIST>

<TYPE REF> ::= <IDENTIFIER> <TYPE RESTRICTION>

<TYPE RESTRICTION> ::= <INPUT/OUTPUT> > |
                      <RANGE>

<TYPE SIZE> ::= ( <EXPRESSION> )

<UNIT ACCESS LIST> ::= <IDENTIFIER>* >

<VARIABLE REF> ::= <IDENTIFIER> <SELECTOR LIST>

<WHEN CASE> ::= Is <NAMELIST> : <STATEMENT LIST>

<WHEN CASE LIST> ::= <WHEN CASE>*

<WHEN ELSE> ::= Else : <STATEMENT LIST>
BIBLIOGRAPHY


