AN ABSTRACT OF THE THESIS OF

James R. Armstrong for the degree of Master of Science in Computer Science presented on August 1, 1988.
Title: Code Generation In The Oregon SpeedCode Universe

Abstract Approved: Redacted for Privacy

Dr. Ted. G. Lewis

The purpose of this project was to create a code generating software tool, which is one section of the Oregon SpeedCode Universe. The Code Generator automatically generates compilable source code to form working Macintosh applications. This source code provides a prototype for the actual final application. This is done with minimal programmer input. The inputs to the Code Generator are: application shell files, a data file with an encoded sequence of actions and a Control List file. The sequences are specified in a Sequence Command file using the sequence command language and define what takes place when a user performs specified actions within the application.The Control List file contains the ID numbers and types of all windows, dialogs, and alerts contained in the resource file plus a count of the number of items in their dialog item lists.

The Oregon SpeedCode Universe is a software tool that provides a system combining a user interface management system with a structured design facility. It is intended for rapid prototyping. The resource file can be generated by the Resource Editor section of the Oregon SpeedCode Universe and the Sequence Command and Control List files by the Graphical Sequencer section.

A literature search provided the theoretical justification for software tools such as OSU and the Code generator in particular. The search also provided a comparative study of other automatic code generating tools produced elsewhere.

The Code Generator is successful. The tool will generate the source code units for a complete Macintosh application. These can be compiled and linked with a binary resource file to produce a working prototype for an application.

Code Generation In The Oregon SpeedCode Universe

by

James R. Armstrong

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed
August 1, 1988

Commencement June 1989

APPROVED:

Redacted for Privacy

Professor of Computer Science, in charge of Major

Redacted for Privacy

Chairman of Department of Computer Science

Redacted for Privacy

Dean of Graduate School

Date Thesis is presented: August 1, 1988.

Acknowledgements

I would like to thank my children, Jason and Rachel, and especially my wife Helen for their patience and kindness during this long process.

Also I would like to thank my advisor, Dr. Ted Lewis, for his help and guidance. Much of this project was done long distance and required extra effort on his behalf. The origination of the concept of the Code Generator and the Oregon SpeedCode Universe rests with him.

TABLE OF CONTENTS

LIST OF FIGURES

CODE GENERATION IN THE OREGON SPEEDCODE UNIVERSE

1) INTRODUCTION

i) BACKGROUND

When computers first came into wide use, the cost of programmer's time was a very small fraction compared to the computer's time and historically there has been great pressure to reduce the cost of computer time. Rapid advances in hardware capabilities and an explosion in the number of machines available has reversed this cost ratio. It is now the programmer's time that is by far the largest fraction of the cost and there is increasing pressure to reduce the cost of programming. This reversal is clearly shown in Figure I.

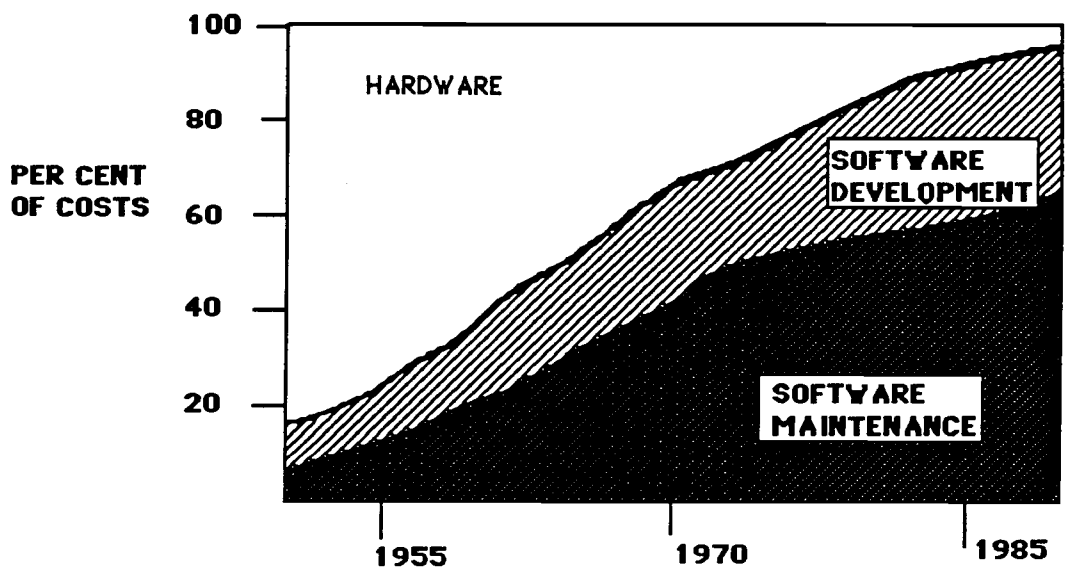

Figure 1. Hardware - Software Cost Trends [Boehm 76]

If the cost of programming is to be reduced, programmers must become more efficient. This can be achieved in several ways: better software tools, automation of the programming process, better methods for design and coding, and increased re-usability of existing code. Also there is a need for better ways of insuring that applications being designed truly meet the user's needs. The Oregon SpeedCode Universe is a rapid prototyping tool designed to

provide all of these advantages for programmers working on the Apple Macintosh computer.

ii) STATEMENT OF PROBLEM

A rapid prototype is defined as a tuple Q = {U,A,F}, where: Q is the resultant prototype, U is the set of user interface objects that can be accessed, A is the set of actions or behaviours defined on the objects in the application and F is a mapping function that creates a graph describing state transitions from one user interface configuration to another. The problem addressed by this work is to generate compilable application source code from a description of Q. More specifically, given a series of events produced by recording user interactions, we want to automatically produce a running prototype application.

iii) LITERATURE REVIEW

The rising cost of programming as a percentage of computing costs has motivated attempts to increase programmer productivity. Software Engineering is a systematic approach to improve the design and construction of applications. These improvements include development methodology and technique as well as software tools [Boehm 76]. The OSU Code Generator is an example of such a tool.

Glickman and Becker focus on potential tools and provide a method for rating both methodologies and software tools. Tools are rated in terms of the maintainability, quality, cost, schedule and productivity they produce. Automatic code and application generators rank as the most useful tools. These are also included at the top of a list of features necessary for a software tool set. [Glickman 85]

Schulz discusses the interaction between software engineering principles, methods and tools, illustrated in Figure 2. A generator for structured programs is one way of directly implementing software engineering principles in a tool. Generation of code and applications

is again considered an important part of a Computer Aided Software (CAS) design system's requirements.

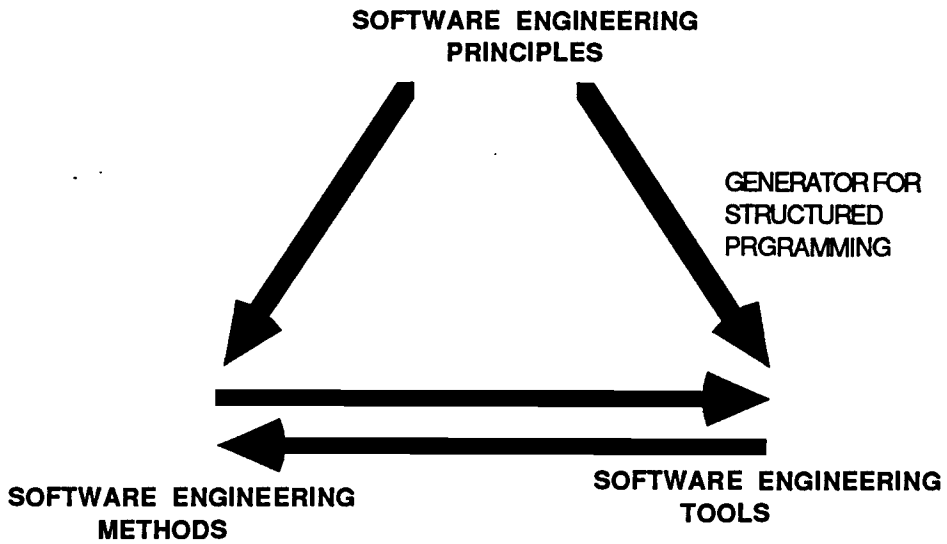

Figure 2. CAS Triangle [Schulz 85]

OSU is a software tool that provides a system combining a user interface management system (UIMS) with a structured design facility. It is intended for rapid prototyping and the production of "vacuous prototypes", which specify an applications user interface but none of its functionality. The addition of more traditional computer aided software engineering (CASE) tools is intended to allow this functionality to be specified, generated and integrated with the user interface to form a "full-fledged application prototype". [Yang 88]

A prototype for an application can be summarized in the formulae Q = {U,A,F}. Q is the resultant prototype. U is the set of user interface objects that can be accessed, in the case of the Macintosh these are such objects as menus, dialogs and icons.The Resource Editor portion of OSU is used to create this [Bose 88]. A is the set of actions or behaviours defined on the objects in the application. F is a mapping function that creates a graph describing state transitions from one user interface configuration to another. These are driven by user interactions and the behaviours of objects in the application. The Graphical Sequencer is is used to partially create A and F [Handloser 88]. The Code Generator produces the Pascal source code to form prototype Q, based on the definitions in U and F.

Essentially this consists of translating F from the intermediate form of the Sequence Command Language into source code. Q is a vacuous prototype. [Lewis 88]

Clearly there is a need for prototyping tools and Code Generators in particular. It is still a very new endeavour but some other efforts have been made in this area. Luqi reports a system that incorporates rapid prototyping and code generation. An intermediate format, the Prototype System Description Language is used to specify the prototype, much like OSU's Sequence Command language. Unlike OSU, the code is generated through the use of re-usable software components. [Luqi 88]

Friman has proposed two prototyping systems: MGEN [Friman 84] and X [Friman 85] . MGEN is a prototyping system for menu driver user interfaces. It uses a "Menu tree" to encode the prototype and can generate the skeleton of an application. It does not generate the source code to implement the specific menus; this is only done in simulation. X is a prototyping system with a format for specifying 'examples' in text format.. X does not seem to generate code, only to provide a method for simulating a prototype.

GIBSGEN is a system designed to generate Fortran programs, primarily of a scientific or engineering nature. It also uses an intermediate language to specify the structure of the application and uses re-usable code modules to generate the application. GIBSGEN requires a great deal of user interaction to produce code, unlike the Code Generator the process is far from automatic. [Bergmark 85]

TUBBS is an object oriented approach to the generation of business applications. It combines existing modules, conceptually similar to the system shells in the Code Generator, with generated modules and hand coded modules. The specifications are data dictionary like definitions and must be done by hand [Van Hoeve 87] .

FRAGTYPES is a programming environment, which supports the programmer by generating some code structures and providing a library of pre-existing code fragments that can be linked in. The process is interactive, not automatic, code generation.[Madhavji 88]

PERIDOT is a prototyping system that seems most like OSU in capability. It can generate code in response to a programmed sequence. It does not support standard user interface items like OSU, these must be built from lower level constructs. However, this means users can create non-standard user interface objects.[Meyers 87]

In a survey of current prototyping technology, Meyers lists several problems with current tools. Two of these can be applied to the Code Generator. First, "too little functionality", the Code Generator only generates the application's framework and specific User Interface code, a vacuous prototype. Future enhancements of OSU may change this. Secondly, "not portable", this tool is restricted to the Macintosh. [Meyers 89]

This Code Generator is unique in many ways. It is fully automatic, requiring no programmer input during the generation process, other than locating input files. This is because the Sequence Command Language was developed to allow the programmer to fully specify the actions that are to be prototyped. The Code Generator then functions as a compiler for a very high level language; generating Pascal not machine source code.

The Sequence Command Language is event driven rather than being based on a control flow. The tokens in the language represent Macintosh resources and the actions they can take. The intent of the generated code is to cause the encoded actions in response to a user's interactions with objects on the Macintosh desktop. This reflects the graphical nature of the interface being prototyped. Since the Code Generator can be driven graphically, the programmer need never know the details of the Sequence Command Language. All of the necessary input files can be produced automatically using the other tools in OSU.

iv) SOLUTION APPROACH

The approach taken to solve this problem was:

- design a Sequence Command Language (SCL) for representing the user interface.

- design Pascal source code templates for the modules that make up a prototype application..
- design and write a Macintosh application to:
    - parse and analyze programs written in SCL.
    -generate compilable Pascal source code equivalent to programs written in SCL. This code generator is original as it functions like a compiler for a very high level language. It generates high level language not machine code and it generates complete applications, not fragments. The language is event driven rather than being based on a control flow. This reflects the fact it is intended to prototype a graphical interface.
    - integrate the generated Pascal code with the module templates to form complete prototype applications.
- integrate the SCL and Code Generator into OSU.

In the remainder of this thesis I will discuss these points in detail. Chapter two contains an overview of the Code Generator's functionality. Chapter three details the implementation of the Sequence Command Language. Chapter four discusses the application templates that were developed. In chapter five the process of parsing the SCL file and generating the Pascal source code is detailed. Chapter six outlines the Code Generator's relationships with the other tools that make up OSU. Chapter seven contains a literature review, the conclusions and a discussion of the Code Generator's limitations and potential enhancements.

v) THE OSU ENVIRONMENT

The Code Generator is one of a set of software tools called the Oregon SpeedCode Universe. The figure below shows the relationship of the various tools that make up OSU.

The Resource Editor lets the programmer design all of the objects that will appear on the desk top and automatically produces the resource file to define them. The resource file can be used as an input to the Graphical Sequencer. This tool allows the programmer to define sequences of events that will occur from user interactions with objects on the desk top. For example the action taken when a menu item is selected. There are two outputs produced from this section: a Sequence Command file that encodes the actions defined, and a Control List file that passes information on certain resource objects to the Code Generator.

The Plum Diagrammer is a section under development. It is intended to allow the programmer to graphically define procedures via Plum diagrams and then automatically generate the source code to implement them. This code would be passed to the Graphical Sequencer for eventual incorporation into the final application[Hsieh 88]. Details of this have not been finalized.

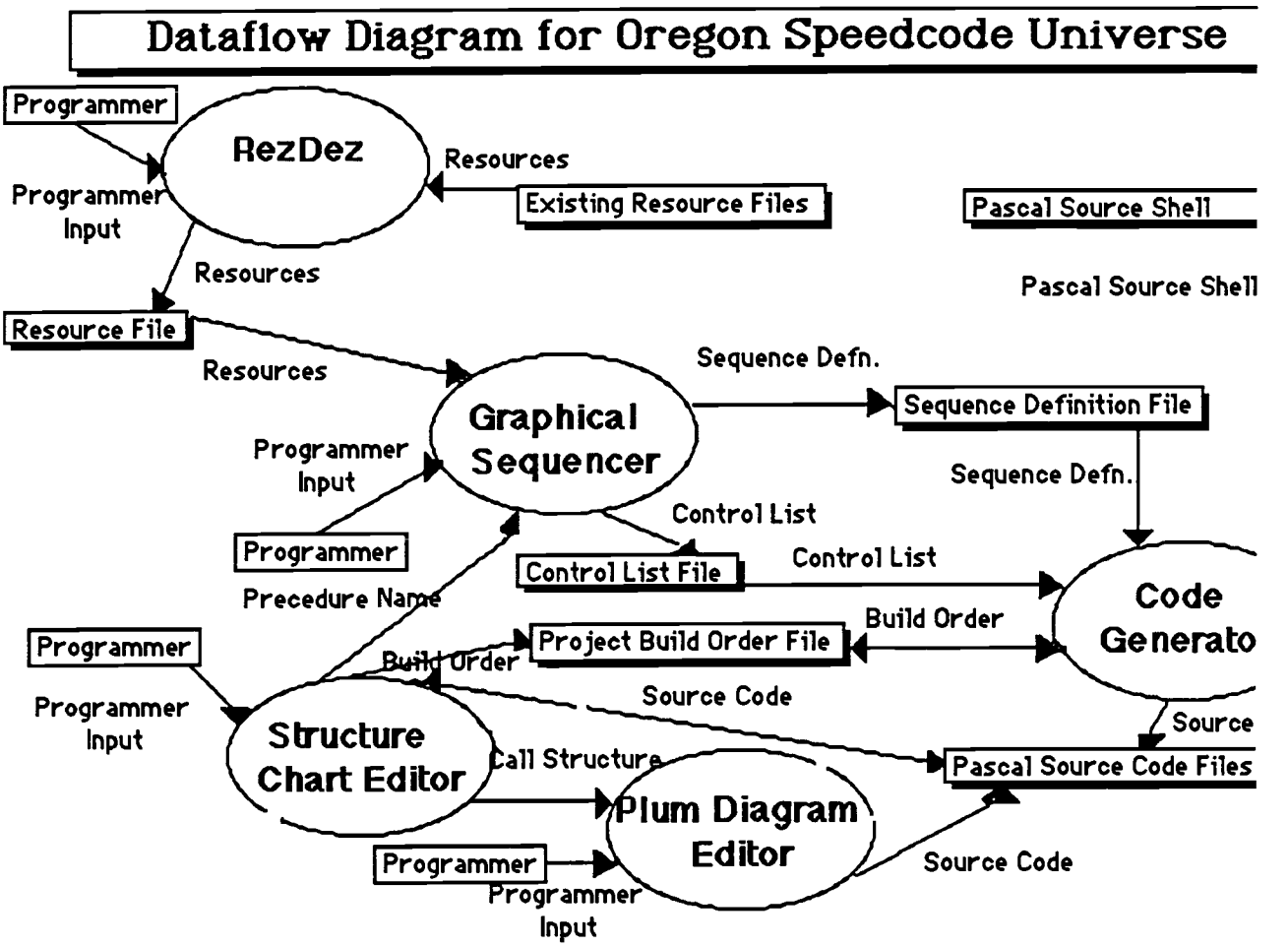

Figure 3. Data Flow Diagram for OSU

The two files produced by the Graphical Sequencer [Handloser 88], the Sequence Command File and the Control List File, are the inputs to the Code Generator, along with shell files that represent templates for units of the final application. The Code Generator creates code based on the actions encoded in the Sequence Command file and incorporates it with the shell files to form the units of the final application. These units, when compiled and linked with the resource file created by the Resource Editor [Bose 88], can be made into a working application.

2) FUNCTIONALITY OF OSU

The function of the Code Generator is to write a program. The data flow for the Code Generator is shown below.

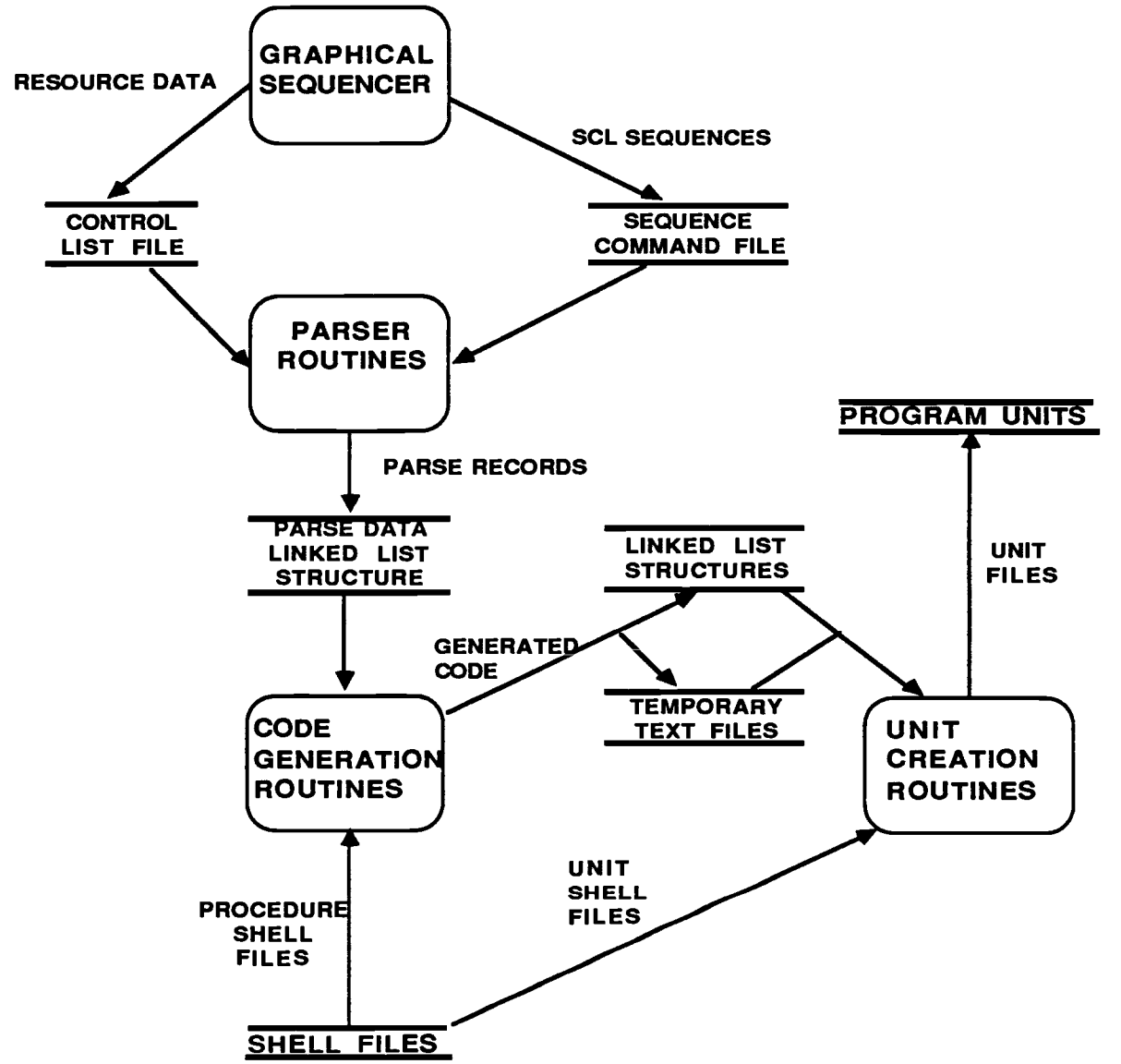

Figure 4. Code Generator Data Flow Diagram

This program is in the format required by the LightSpeed Pascal compiler, and designed to run on the Apple Macintosh computer. The inputs to the program are: a Sequence Command file generated by the Graphical Sequencer, a list of all the Dialogs, Alerts and

Windows in the resource file with a count of the number of objects they contain, and a set of shell files.

The Code Generator first parses the Sequence Command File and creates a linked list data structure to encode the information obtained during parsing. This linked list will be referred to as the parse stack. If a syntax error is detected, an error message is displayed to the user and the program returns to the top level. If no error is detected, the program proceeds to process the parse stack it has created. The first step is to find the dialogs and alerts that need procedures generated to handle the actions of their objects. This is done by finding the sequence items that are associated with dialogs or alerts. These are INSERTSTR and ITEMHIT = DITLITEM. Code is generated to perform the action requested and it is inserted into the appropriate procedure. If that procedure does not exist, it is created.

Once the procedures and their associated sequence items have been processed and removed, the remainder of the parse stack is processed in three parts. First any ITEMHIT = GOAWAY sequences are processed. These allow actions to happen when a window is closed. Then code is generated to handle the sequence action in the Initialization section. These are sequences that occur when the program first starts, before the user has a chance to take any actions.

Finally the menu block is processed. This is the section that allows the user to encode a tree like sequence of events, with the menu lists acting as the roots. This can also result in the generation of procedures, one for each menu item that has sequence actions associated with it. As sequence statements are processed, their records are removed from the parse stack. If an error is detected during processing, an error message is displayed for the user and the program returns to the top level.

The code that is generated during this phase is stored in linked list data structures or temporary text files. Once the processing is complete, the parse stack is gone, and the actual program is created. This is done by reading a series of fourteen shell files, each of which contains some meta symbols. The original shell file is output with a new name, the

meta symbols removed and generated code, if any, inserted in their places. There is also a text file that lists the file names for the new units and their hierarchical order.

The original shell files are simply read, and are left unaltered at the end of the process. As the three steps of the Code Generator are under way, messages appear to inform the user of progress. The only other potential interaction with the user is to ask for the location of input shell files if the program cannot locate them. The user is informed the file is missing and then it is searched for via a standard Macintosh SFGetFile function call.

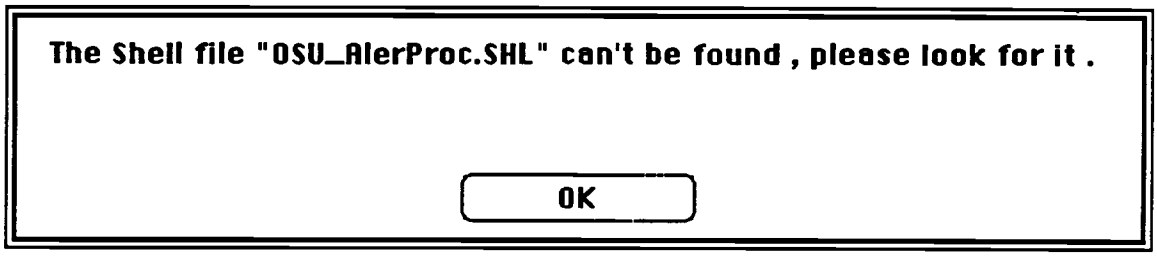

Figure 5. Missing File Dialog

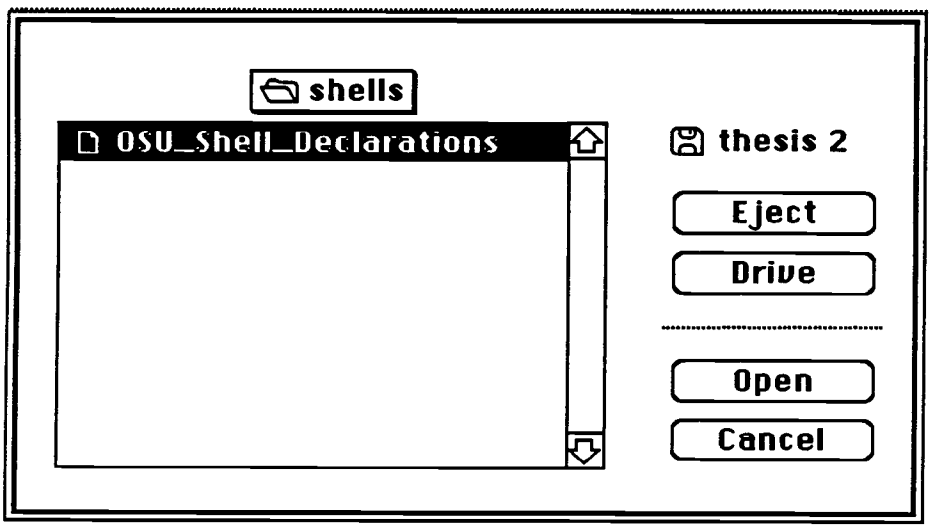

Figure 6. Standard SFGETFILE Dialog

The only other interactions are the messages displayed if errors are detected or the user does not locate a needed shell file. Any of these conditions, or a successful completion of the code generation, cause the program to return control to the top level. At this point the user can exit from OSU, enter a different section or re-enter the Code Generator. The user can take the code units produced, and based on the build order listed in the Project file, compile a working program using LightSpeed Pascal

3) SEQUENCE CONTROL LANGUAGE

i) BACKGROUND

The intent of the Sequence Command Language is to allow the programmer to define sequences of events deriving from user actions, and then have code generated to implement them. The preliminary version of the language was a subset of the functions now supported in the syntax. One of the major changes has been in the amount of information encoded in some of the sequences. This information was added as it was found to be necessary for code generation. The reason for this is a decision that the Code Generator would not have direct access to the Resource file and thus any information must be either encoded in the sequence or passed as a parameter to the Code Generator. The current program passes a list of the Resource file's dialogs and alerts plus a count of their 'control' items. Any other information needed is encoded in the language.

There are some functions supported in the Code Generator, but not in the Graphical Sequencer. These deal with Window, Picture and Icon objects [Handloser 88]. A programmer can implement these functions by editing the Control List File, after the Graphical Sequencer has created it, before passing it to the Code Generator. Another decision made during development was to allow the programmer to modify or even create a Sequence Command file outside of OSU. This meant adding a great deal of error checking to the parsing of the Sequence Command file. If it was always generated automatically then the format could be expected to be syntactically correct. The syntax was not appreciably changed to support this decision and the language is not very user friendly. Comments and blank lines were added to allow a programmer to comment and format the file. The number of blank spaces at the front of a line and rigid indentation of nesting were not enforced.

ii) USAGE and STRUCTURE

The Sequence Command Language has the general format: <verb> <object> <[symbol]>. All the verbs except two have an object and all the verb - object pairings plus one of the two single verbs have symbols. Some examples are:

```
CHANGE CURSOR [ARROW] ;
CLOSE DIALOG [324] ;
DO [SORT] ;
QUIT ;
```

The verb ITEMHIT is similar to a 'begin' statement in Pascal, it signals the start of a block of sequence instructions that are grouped. The object for the ITEMHIT verb tells what kind of a block and some of its objects have a symbol associated with them. These groupings of sequence instructions are ended with a line containing a single period.There can be a number of other sequence statements enclosed within the block. This is similar to an 'end' statement in Pascal. Some examples:

```
ITEMHIT = DITLITEM [1=D212=R] ;
        CLOSE DIALOG [324] ;
        QUIT ;
. ;

ITEMHIT = INIT ;
        CHANGE CURSOR [ARROW] ;
. ;

ITEMHIT = MENUITEM [3=2] ;
. ;
```

Every sequence instruction has a space and a semi-colon at the end of the line. Comment lines can be included by making the first non-blank character on a line an asterisk, this causes the rest of the line to be ignored by the Code Generator. Totally blank lines can also be included and are also ignored.

The sequence is always communicated to the Code Generator in the form of a Sequence Command File. This file is used as the main input to the Code Generator and must be created using the syntax of the Sequence Command Language. This file is automatically created by the Graphical Sequencer if the programmer chooses that option. As it is only a text file, it could be created by a programmer with any word processing program. It is also

possible to modify a Sequence Command file created by the Graphical Sequencer. In either of the latter two cases, the programmer must very carefully adhere to the syntax of the sequencing language, detailed in Appendix i.

This file always has two sections, an Initialization block and a MenuBar block. These are ITEMHIT blocks and may contain nothing. An example of a short Sequence Command file is shown in Figure 7, a more detailed example is in Appendix iv.

```
* this is a sample Sequence Command File
ITEMHIT = INIT ;
        OPEN DIALOG[212] ;
. ;
ITEMHIT = MENUBAR ;
        ITEMHIT = MENUITEM [3=0] ;
                ITEMHIT = MENUITEM [3=1] ;
                        DO [SORT] ;
                        DISABLE MENUITEM [3=4] ;
                . ;
                ITEMHIT = MENUITEM [3=2] ;
                        QUIT ;
                . ;
        . ;
. ;
```

Figure 7. Sequence Command File

The Initialization block contains sequence statements that generate code to implement an application's initial state. Most commonly this block would: set the initial cursor state, set up a disable or enable pattern for menu items, and display an initial dialog or alert. If this dialog or alert has parameter text or controls, which are called ditl items in the language, then these could also be set to their initial values. Other actions could occur at this time but these are the most likely. This block cannot contain all types of sequence statements, for example no menu list or menu item blocks can be contained in this section.

The menubar block has no restrictions on which items can occur within it, except ITEMHIT = INIT, but it is restricted in format. All code items must be associated directly or indirectly with a menu list or item. There are itemhit blocks for menu lists, within which can be nested itemhit blocks for the menu items contained within that menu list. Within

these menu item blocks can be any verb object pairings except other menu item or menu list blocks. Also code can be included which is associated with a dialog's or alert's ditl items. Sequence statements that affect the ditl items or parameter text can appear in any menu item block. Code generated for these sequences is inserted directly into the procedure generated for the dialog or alert they are associated with. Typically there would be a call to this procedure generated in the menu item blocks. Example:

```
ITEMHIT = MENUBAR ;
        ITEMHIT = EDIT [3=0] ;
                ITEMHIT = UNDO [3=1] ;
                        OPEN DIALOG [203] ;
                        ITEMHIT = DITLITEM [2=d203=c] ;
                                CLOSE DIALOG [203] ;
                        . ;
                . ;
        . ;
. ;
```

This example shows a simple menubar block with one menu list block, EDIT, containing one menu item block, UNDO. Thus if the user selected the item 'UNDO' in the 'EDIT' menu, dialog number 203 would be opened. The DILTITEM block is for a ditl item in dialog 203, thus forcing the creation of a procedure to handle the dialog. Inside that procedure, if DITLITEM number 2 were selected the dialog would be closed, the procedure would terminate. The OPEN verb sequence would have a procedure call generated for it, since dialog 203 is a procedure.

The objects represented in the language are those that are most commonly used in a Macintosh user interface. The verbs were chosen to support a wide a range of operations. The menu lists and menu items are an integral part of virtually all Macintosh applications. This is reflected in the Sequence Command file, which has one of its two sections devoted solely to the consequences of the user selecting menu items and menu lists. The ITEMHIT verb is used to indicate a block of statements that should be implemented if the user selects a specific object. This object could be a menu list, a menu item, a ditl item inside a dialog or alert, or the goaway box in a window. The ITEMHIT verb is also used to mark the two

format blocks within a Sequence Command file. By itself this verb has no effect on code generation, it exists only to indicate where the code generated for the sequence statements contained within its block should be located. This code reflects the actions that are to occur if the user selects the object specified by the itemhit statement.

Other verbs which can be used in conjunction with menu lists or items are: CHECK, UNCHECK, ENABLE, and DISABLE. The first pair of verbs affect only the appearance of the menu items. The second pair change both the appearance and function of the menu lists and items, by making them active or inactive. These four verbs can also be used with ditl items inside dialogs or alerts. The ditl items are affected by these verbs the same way as the menu lists and items. Examples:

```
ENABLE EDIT [2=0] ;
DISABLE UNDO [2=1] ;
CHECK DITLITEM [4=A224=2] ;
UNCHECK DITLITEM [4=A224=2] ;
ENABLE DITLITEM [1=D212=1] ;
DISABLE DITLITEM [2=D212=1] ;
```

Another set of objects common to virtually all applications are: windows, dialogs, and alerts. All of these are types of rectangular areas that can be opened on the screen to communicate information to, or elicit responses from, the user. Verbs common to these are OPEN and CLOSE, which cause the actions one would think. Also FRONT and BACK which can be used with dialogs or windows. It is possible to have a number of windows or dialogs present on the screen at once, with only the frontmost one active. These verbs make a specified window or dialog the frontmost, or move it to the back. They are not used with alerts, since by their nature alerts reflect actions that require immediate attention and must be dealt with before any other actions can continue. Examples:

```
OPEN DIALOG [203] ;
CLOSE DIALOG [SFPUTFILE] ;
FRONT DIALOG [201] ;
BACK DIALOG [202] ;
OPEN ALERT [333=1] ;
```

Dialogs and alerts can have a list of ditl items associated with them. This list can represent controls, messages, edit boxes or a number of other types of items. These lists can be controlled using the ITEMHIT, CHECK, UNCHECK,ENABLE, DISABLE, and INSERTSTR verbs. The first five have been discussed already, the last is used to load a string into a variable 'message'. These 'messages' are called static text blocks and they can have either a fixed message or be flagged as paramtext. This means they can have messages inserted into them during run time. Any given dialog or alert can support up to four paramtext messages. The verb INSERTSR is used to insert a predefined string constant from the resource file into the paramtext message of a dialog or alert. This is a very useful feature for a programmer as it can allow a single dialog or alert to be used to display a wide variety of messages to the user. Example:

```
INSERTSTR 2 [17] ;
```

Another feature common to all applications is the cursor. The verb CHANGE can be used to alter the cursor's appearance. It can be changed to any of five predefined shapes as well as made invisible or visible. Two other objects supported by the language, pictures and icons, are not used as commonly. Pictures are graphical images that are displayed within a bounded rectangle. The verbs ADD and TAKEOUT are used to create this rectangle and display the picture, or remove the picture and the rectangle. Icons are small graphical images that can be used in the place of controls or menu items. The verb PUTIN can be used to insert icons in the place of controls or menu items. Examples:

```
CHANGE CURSOR [IBEAM] ;
ADD PICTURE [115=20=20=100=200] ;
TAKEOUT PICTURE [115] ;
PUTIN 124 [A=224=3] ;
```

The verb QUIT is not associated with any objects. It indicates an action will cause the application to terminate. Commonly a menu item named quit is included and its action is simply to terminate the application. Without this verb it would be impossible to generate applications with proper exit points. This is the only verb that does not have a <symbol>

and one of only two without an <object>. The verb DO has no <object> but its <symbol> is the name of a procedure. This verb allows the programmer to insert calls to procedures unique to that application into the generated code. Examples:

```
DO [QUICKSORT] ;
QUIT ;
```

The symbols for the other verb - object pairings encode information for that specific usage. Most commonly this includes the ID numbers of the objects involved. In the case of some verb - object pairings there is more information that must be included. Specifics of this plus further detail on use of the sequencing language generally, can be found in Appendices ii and iii.

iii) RESTRICTIONS

It is important that the syntax be carefully adhered to if editing a Sequence Command file. It was primarily designed with automatic creation and parsing in mind, not programmer editing. It is important that all sequence statements end with a blank and a semi-colon. This is used by the parsing routines to detect the end of the statement. Most errors will be caught by the Code Generator in the parsing or code generation phase, but it still could be an annoyance.

There is no guarantee that generated applications will be syntactically correct, and if compilable, will be logically correct. The code lines that are generated will always be correct in and of themselves, but syntax errors may be present in the program as a whole. For example the names given for user procedures could be reserved words in Pascal. Example:

```
DO [BEGIN] ;
```

This would generate the code line 'BEGIN;', a syntax error in Pascal. It is also possible to create sequences that will produce run time errors in the final application. It is possible to create infinite loops or applications without exits. Example:

```
ITEMHIT = MENUBAR ;
        ITEMHIT =  EDIT [3=0] ;
                ITEMHIT =  UNDO [3=1] ;
                        OPEN DIALOG [201] ;
                        ITEMHIT = DITLITEM [1=D201=c] ;
                                OPEN DIALOG [201] ;
                        . ;
                . ;
        . ;
. ;
```

Figure 8. Sequence Error Example

This would generate a procedure, 201, which can only call itself. There would be no exit from this procedure or from this application. An effort has been made to detect as many errors as possible but this is not a compiler. It is assumed that programmers will have some knowledge of Pascal and structured programming techniques.

4) SHELL FILES

The key to the Code Generator's ability to produce the source code for a complete working program is the use of shell files. These are text files that effectively contain the template of a program unit with some special symbols added. These symbols, called meta symbols, indicate the locations where generated code may be inserted. Some meta symbols will always be replaced with code, the rest are done conditionally based on the contents of the Sequence Command file. The meta symbols consist of a percent character followed by a digit from 0 to 9 (%0 - %9). The percent character appears no where else in the shell files.

i) HISTORY AND DEVELOPMENT OF THE SHELL FILES

The original 'MacGen' program from which OSU has evolved used a single Pascal shell file. This contained a large number of meta symbols to be replaced with code generated by the program [Lee 86]. It was planned that most of the code generated would be stored in text files and added to the main shell program using Pascal's include mechanism.

When OSU was developed, it was necessary to revise the shell file. The original Pascal shell was divided into four files. These form the basis of a generic Macintosh application and contain no generated code other than unique names for the units themselves and any units named in their uses clause. The rest of the shell files are used only for code generation and were developed for the Code Generator.There are ten shell files used to create output units. Figure 29 shows all the shell files that make up a project, in their build order.

```
1 : OSU_ Declarations
2 : OSU_ Globals
3 : OSU_ SystemCalls
4 : OSU_ UserProcedures
5 : OSU_ SimpleAlert
6 : OSU_ SimpleDialog
7 : OSU_ GroupN
8 : OSU_ MenuProc
9 : OSU_ MenuCase
10 : OSU_ GoAway
11 : OSU_ Initialize
12 : OSU_ Procedures
13 : OSU_ MainEvent
14 : OSU_ Main
```

Figure 9. Unit Shell Files in Build Order

Also it was necessary to have templates for procedures that are created. There are three of these, one each for dialogs and alerts plus a generic procedure template used for menu item or user generated procedures. This last shell file is shown in Figure 10. The processing of the shells is discussed in section five - iii.

```
PROCEDURE %1;
    %2
    BEGIN
        %3
    END; { %1 }
```

Figure 10. Sample Procedure Shell File

ii) SYSTEM SHELL FILES

The four system shell files are: OSU_Declarations, OSU_Procedures, OSU_MainEvent and OSU_Main. These are processed outside the Code Generator and do not have generated code inserted in them. The first of these, OSU_Declarations, contains global declarations that are always needed by the application. Any global declarations created during code generation are included in a Code Generator shell. The second system shell, OSU_Procedures, contains the utility routines needed by the generic application. There are

routines here to do the application's initialization, perform some generic window and dialog handling, and display messages to the user. These are routines that are always needed by an application, regardless of what the user encodes in the Sequence Command file.

The third file, OSU_MainEvent, contains the main event loop. It is a code loop that continually checks for input to the user interface via the keyboard or a mouse input on the desk top. The main event loop then passes this information and control to the appropriate routine. For example if the user selects a menu list with the mouse, control is passed to a toolbox routine that displays that list's menu items. If in turn one of them is selected, then a routine in the menu shell is called. If code has been generated for that menu item then a procedure containing that code will be called. This may in turn open a dialog with controls and potentially continue in a long sequence of events. When the user exits from this sequence, control passes back to the main event loop. The main event loop also passes input information to active desk top objects, such as a control hit within a dialog. This information is needed by the procedure controlling that dialog.

The last shell, OSU_Main, merely contains a call to the initialization routine and to start the Main Event Loop. It is illustrated in Figure 11.

```
{------------------------------------------------------------------------ }
{   Copyright 1988, Oregon State University.                              }
{   This file is the "Main program" File for the Application.             }
{   This file was produced using the Oregon SpeedCode Universe            }
{  Prototyping system.                                                    }
{-------------------------------------------------------------------------}

PROGRAM %1.MAIN ;

USES
        %2;

        BEGIN
                SetUp;
                MainEventLoop;
        END. {%1}
```

Figure 11. Sample System Shell File

iii) SEQUENCE SHELL FILES

There are ten unit shell files used by the Code Generator to create the output application. These are shown in Figure 12.

OSU_ Globals
OSU_ SystemCalls
OSU_ UserProcedures
OSU_ SimpleAlert
OSU_ SimpleDialog
OSU_ GroupN
OSU_ MenuProc
OSU_ MenuCase
OSU_ GoAway
OSU_ Initialize

Figure 12. Sequence Shell Files

OSU_Globals contains any global declarations, constants or variables, generated during code generation. OSU_SystemCalls contains a set of procedures that provide an interface between the operating system's toolbox routines and the code generated. The Code Generator creates calls to the procedures in this shell, rather than directly to the toolbox. This encapsulates all toolbox calls in one unit and allows for easier upgrading or maintenance in the future.

OSU_UserProcedures contains those procedures generated in response to sequence statements with the DO verb. These are procedures unique to the user's application and consist only of empty stubs. Calls to these procedures are generated in the application and the user can later add code to implement the procedure's actions. OSU_SimpleAlert and OSU_SimpleDialog contain those procedures, of their respective type, that are 'simple'. Simple procedures are those that do not make calls to other generated dialog or alert procedures. They may make calls to procedures in either the OSU_UserProcedures or OSU_SystemCalls units.

OSU_GroupN is used to create the dependency group units. These contain dialog and alert procedures that are not simple and do make calls to other generated dialog or alert procedures. Each dependency group that is detected is output as a separate unit using this

shell. A unique name is generated for each of these. This is the only shell file that can be used to generate more than one output unit. Procedures included in these groups may make calls into any of the units already discussed, but not into any other dependency group.

The OSU_MenuCase and OSU_MenuProc shells contain the code to handle menu lists and items. The menu case shell contains code to handle any menu lists or menu items specified in the Sequence Command file that the user may select. The menu items, if selected, call a generated procedure. These procedures are contained in the MenuProc shell and can call procedures in any of the units already discussed.

OSU_Init shell contains code to implement the actions specified in the initialization section of the Sequence Command file. This shell contains a function that is called whenever the application starts up and the initialization code generated is inserted into this procedure. OSU_GoAway contains a procedure that is called when a window is closed by the user selecting its goaway box. The procedure implements a set of actions specified in the Sequence Command file. Regardless of any actions specified, this procedure will close the window whose goaway box was selected. It is possible for code generated for either of these two shells to include calls to procedures included in any of the other sequence shells except the two menu units. The OSU_Init shell is shown in Figure 13.

```
{-------------------------------------------------------------------}
{   Copyright 1988, Oregon State University.                        }
{                                                                   }
{ This file is the  UNIT for the Sequence  Initialization code.     }
{ The UNIT OSU_GLOBALS is used, and it uses all of                  }
{ the other generated units that are needed.                        }
{-------------------------------------------------------------------}
UNIT %1;

INTERFACE

USES
        %2;

        PROCEDURE Seq_Init;

IMPLEMENTATION

PROCEDURE Seq_Init;   {no parameters}
        BEGIN
                %3
        END;

END. {%1}
```

Figure 13. Sample Sequence Shell File

iv) PROCEDURE TEMPLATE SHELL FILES

There are three shell files used as templates in the generation of procedures: OSU_Dialproc.Shl, OSU_AlerProc.Shl and OSU_Proc.Shl. These are used to generate procedure files for dialogs, alerts, user procedures and menu items. These procedures are written out to temporary text files when processed and later read into the SimpleDialog, SimpleAlert, MenuProc or UserProcedures units. The dialog and alert shells may also be used to create procedures for inclusion in the Depend Group unit. OSU_Proc.Shl was shown in Figure 10.

5) DETAIL OF FUNCTIONALITY

At the top level of the Code Generator there is a single procedure, Seq_Control, that is called by OSU to initiate code generation. This procedure calls a procedure to parse the Sequence Command file, and another one to process the stack created by the parsing routine and to produce the Pascal code units. In between these two calls are checks for several error conditions. If a fatal error is detected here or elsewhere, the Code Generator returns control to the main body of OSU, and displays an error message to the user.

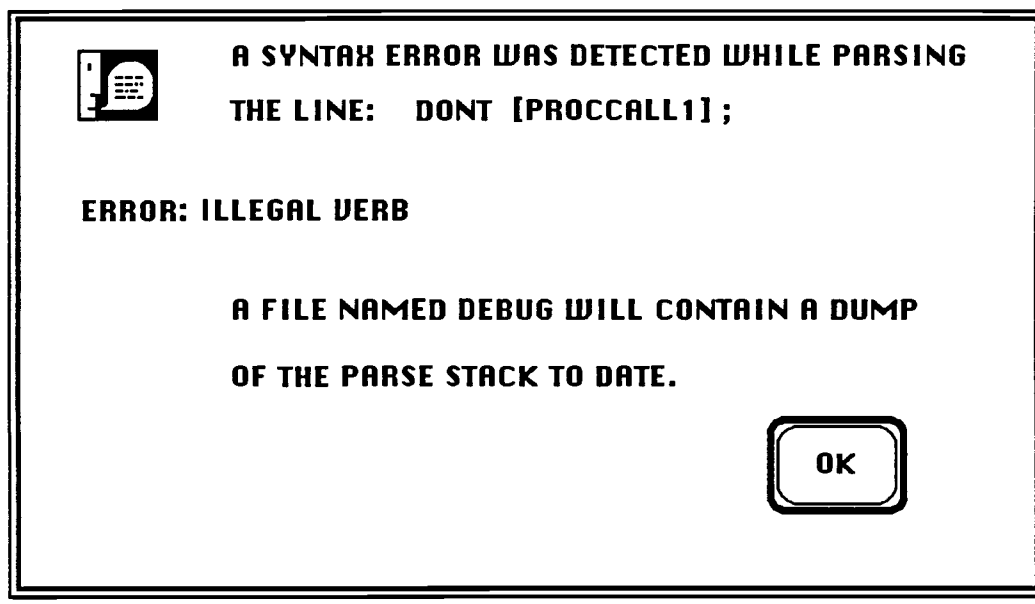

Figure 14. Sample Error Message Dialog

If no errors are found then this routine returns the desk top to its original state, and returns to OSU's desktop. Within the code generation there are three distinct phases.

i) PARSING DETAIL

The Code Generator parses the Sequence Command File and encodes the information from each statement as a record in a linked list of parse records, the parse stack. A modeless dialog is displayed to inform the programmer that the Code Generator is parsing.

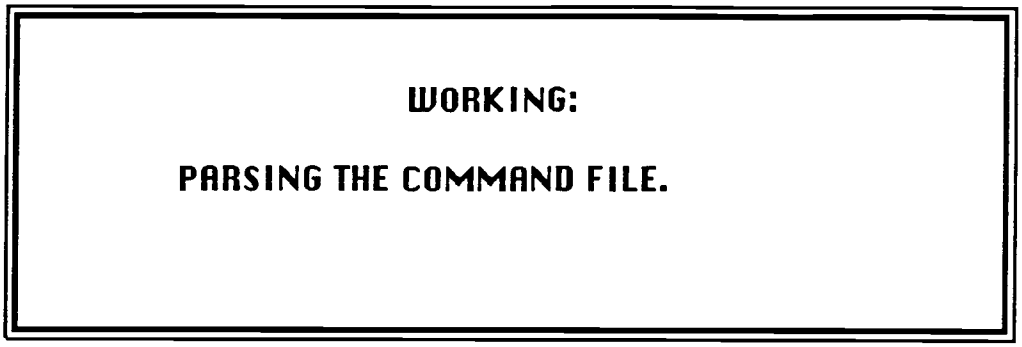

Figure 15. Parsing Dialog

The parse records contain fields to encode the information from the sequence statement plus fields to encode information determined by context during the parsing process. The verb and object are encoded numerically, the symbol is stored as is and is interpreted during the code generation phase. Another field is used to match itemhit and period pairs. The last field stores the statement type. It is possible for the last two fields to be changed due to information detected while parsing later sequence statements; the others are never changed from their initial values.

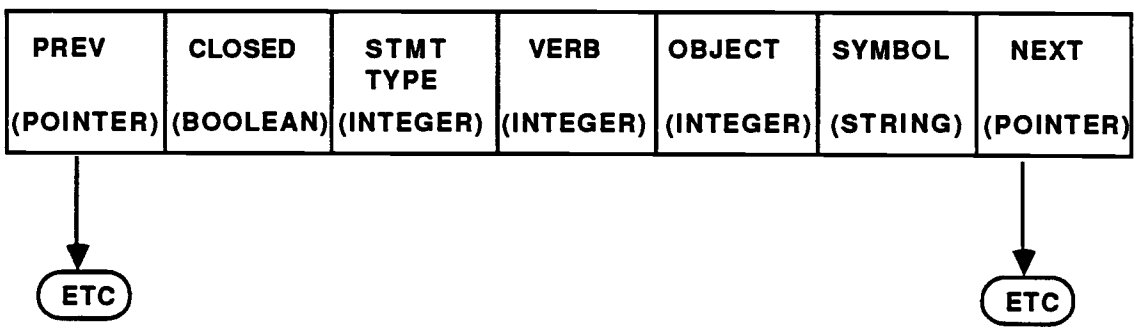

Figure 16. Structure of a Parse Record

The Sequence File is opened as a text file and read from the beginning, line by line. A check is made for blank lines, otherwise the line is passed to the parsing routine. If a parse error is detected, the program will display the appropriate error message and return control to the top level of OSU. If no errors were found then the parse stack is passed to the code generation phase. A large number of potential errors are checked for during parsing due to

the decision to support programmers modifying or creating Sequence Command files outside of OSU.

The line parsing routine decomposes a line into its component parts, checks that each part is a legal member of the expected type and determines the values to store in the parse record for that sequence statement. If no errors were found, a new parse record is created and the values loaded into the appropriate fields. The record is then added to the bottom of the parse stack. Default values are set for all fields so that sequence statements that do not have all parts will have legal values for all fields. For example the QUIT verb has no object or symbol, so default values will occupy those fields.

The line is first checked to see if it is a comment line and if so, the rest of it is ignored. If not, the first component in a line is expected to be the verb. The verb is checked against a list of legal values. This check returns an index to a code block that processes the verb. Most merely set the verb field value to the proper numeric code. The QUIT verb has no other components so the rest of the parsing process is skipped. The ITEMHIT verb also sets the field 'closed' to false. When the matching period for this ITEMHIT is found, the closed field is reset to true. This is how the final check is made that all ITEMHIT verbs had a corresponding period, any fields left false indicate an error.

The next step for all of the verbs, except QUIT and periods, is to obtain the second component from the line, normally the object. In the case of the INSERTSTR and PUTIN verbs the second component is not a normal object and in the case of the DO verb there is no object, just a symbol. These three verbs deal with the rest of their lines separately. For the rest of the verbs there is an object and it is obtained and checked in the same way as the verbs were. The return value is used as an index to a second code block to process the objects. The object - verb pairing is checked to see if it is a legal combination and the rest of the line is processed. If a symbol is expected, it is obtained from the line. Some symbols or parts of symbols are checked for range errors or to insure they are valid for that verb -

object paring. For example there are only seven legal symbols that can be used with CHANGE CURSOR. Any errors detected at this point are dealt with as noted previously.

Several of the objects require some unique processing. In the case of the objects INIT and MENUBAR, flags are set when they are processed. These flags are later checked to insure both were present in a file and are also used to check that more than one of each does not occur. The USERDEFINED objects, which are the names of menu items or menu lists have an underscore character appended. This is to avoid definitions that conflict with reserved words in Pascal. For example the word 'file' is a common problem. Also these objects are scanned to detect occurrences of the Apple symbol. This is a character that is commonly used as the title for the first menu list in an application. This is normally a non-printable character and it would cause problems in code generation. It is replaced with the string 'AppSym'.

If the object case has been successfully processed and no errors have been detected to that point, then a parse record is created. Blank lines and comment lines have no records created for them. The parse stack is a doubly linked list to allow traversing it in both directions. This is necessary in the code generation phase when records can be pulled out of the stack at any point. There is a one to one correspondence between records in the parse stack and valid sequence statements in the Sequence Command file. When the parsing is completed, and if no errors have been detected, the completed parse stack is passed to the next phase, code generation.

ii) CODE GENERATION DETAIL

The Code Generator closes the Parsing dialog (Figure 15) and displays a dialog to inform the programmer that code generation is now under way.

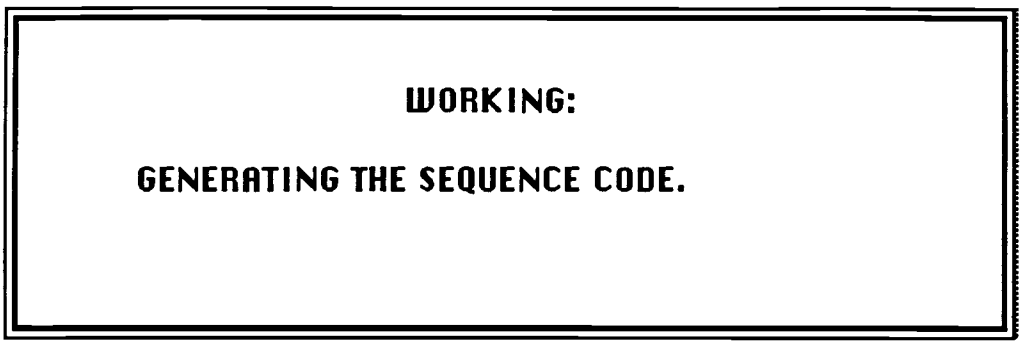

Figure 17. Code Generation Dialog

The code generation phase of the application has two objectives. First the creation of lines of Pascal code for the various verb - object pairings encoded in the parse stack by the parsing routine and secondly the storage of the code. This stored code will be accessed during the unit creation phase and inserted into the output units. The first objective is handled by repeatedly scanning the parse stack, each time looking for sequence instructions that will be grouped together in the output. As these are found and processed, the code they generate will be added to a temporary text file or linked list data structure. The code generation can range from creating a single line of code to an entire procedure. The first thing done in the code generation phase is to initialize the linked list structures that will be used to store any procedures generated, and to process the Control List file passed as a parameter into the top level of the Code Generator. This file contains the ID numbers of all dialogs and alerts in the resource file plus a count of the number of items in their Ditl lists. This file is read in and the information in it stored so it can be consulted whenever a new procedure is to be created.

The parse stack is always scanned from the bottom up. The main reason for this is to circumvent problems with nesting of blocks. Itemhits on Goaways or Ditl Items can be nested. If the stack is processed from the bottom up then the inner most block in a nested group is processed first, then the next outer group and so on. Also the relative location of

INSERTSTR sequences to dialog or alert OPENs is important and this maintains those relationships.

The first thing the the Code Generator needs to determine when processing the parse stack is which dialogs or alerts, specified in the sequence, will need to have procedures created to handle the actions specified for their ditl items. Not only does this necessitate the generation of the procedure, but it affects the code that is generated for any OPEN or CLOSE sequence statements associated with that dialog or alert. If a procedure is created, then the OPEN sequences will result in a procedure call; otherwise simply a call to the generic procedure in the SystemCalls unit. If a generated procedure is involved, it is illegal to use CLOSE sequences for that dialog or alert, except in sequence instructions that generate code to be located within the procedure. It is not sensible to attempt to 'close' a procedure from outside itself. Inside the procedure, the code generated for a CLOSE verb results in the procedure being terminated and control returned to the calling sequence.

The first thing done is to make a list of all the procedures that will need to be created. This is done by scanning the parse stack for sequence instructions that indicate a dialog or alert will need a procedure generated for it. The verb - object pairings that indicate this are: INSERTSTR , ENABLE DITLITEM, DISABLE DITLITEM, CHECK DITLITEM, UNCHECK DITLITEM and ITEMHIT = DITLITEM. When one of these is found, the ID of the dialog or alert it goes with is obtained by decoding the symbol. Once a dialog or alert is identified for procedure creation, a check is made with a list of procedures already created, to avoid duplicate definitions. If it does not exist, then it is a new procedure and several things occur.

First a check is made with the information from the Control List file. This prevents the creation of procedures for dialogs or alerts that are not present in the Resource file. That would be a fatal run time error and is dealt with as an error. If the dialog or alert is found, then the number of items in its Ditl list is obtained. Then a unique name is created for the procedure, and it is added to the list used to prevent the creation of duplicate procedures. A

record structure is created to store the code that will be generated for this procedure. It is illustrated in the figure below. There is a header record for each procedure plus a variable list of code list headers.

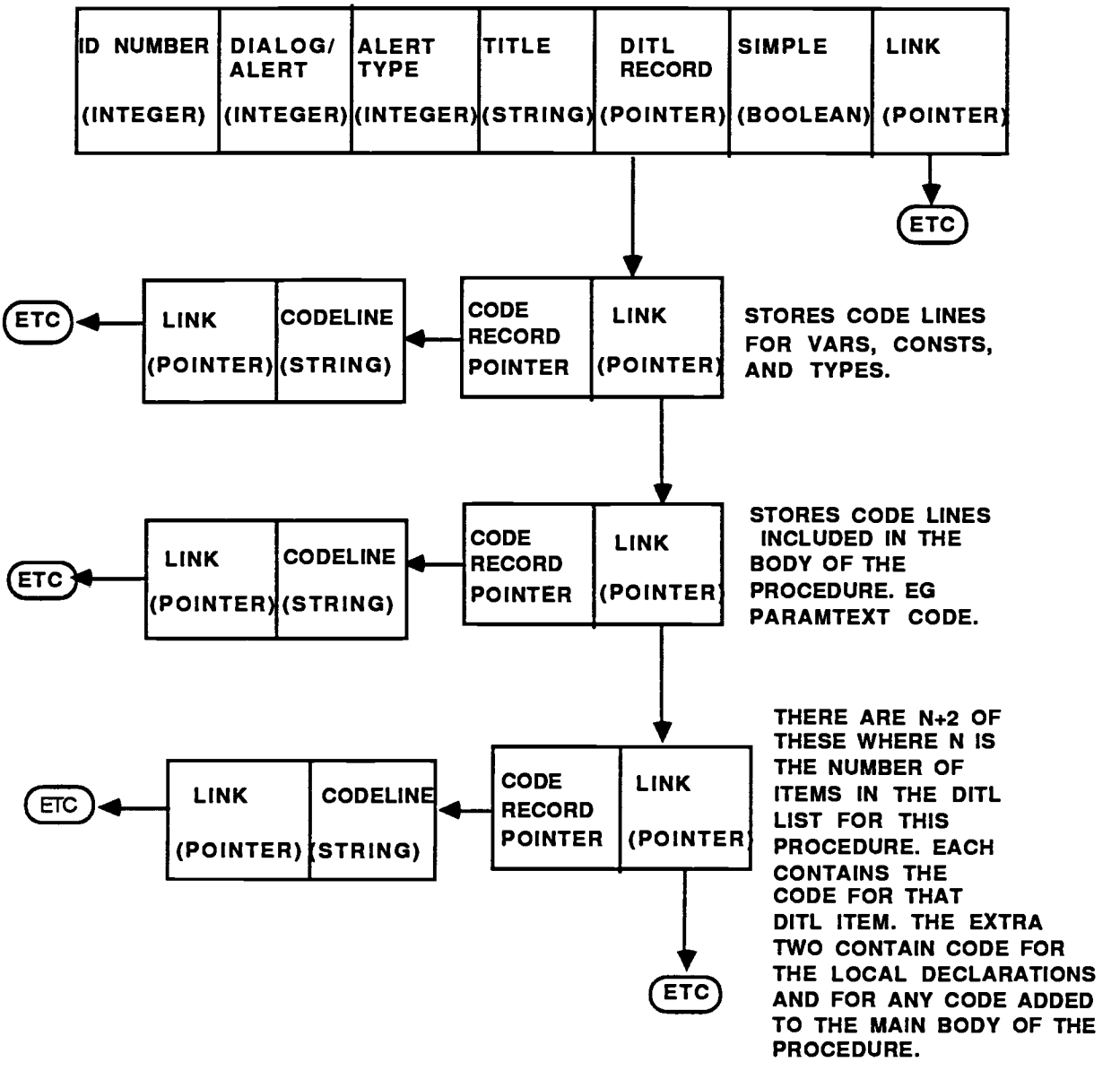

Figure 18. Procedure Record Structure

After the structure is created, all OPEN or CLOSE sequences for that dialog or alert anywhere in the stack are flagged as being associated with a generated procedure. Any OPEN or CLOSE verbs not flagged will be dealt with as generic dialogs or alerts.

The DO verb also causes procedures to be created. Again checks are made to prevent duplicate definitions.Figure 19 shows a sample procedure created for a DO verb sequence.

```
PROCEDURE PROCCALL1 { };
    BEGIN
        {USER CODE INSERTED HERE}
    END; { PROCCALL1 }
```

Figure 19. Sample of a Generated User Procedure

Once all procedures have been created, then the code that will be included in them is generated.This is done by processing all of the verb - object parings that caused the procedures to be created.

The INSERTSTR verbs are done first. These cause strings defined in the resource file to be inserted into text items defined as having paramtext. As a record from the parse stack is selected for processing, it is removed from the stack. INSERTSTR sequence instructions are, by definition, associated with the first dialog or alert for which an OPEN sequence is found, above the INSERTSTR sequence in the stack. In practical terms this means that dialogs or alerts have to be opened in the Sequence Command file before their INSERTSTR sequences occur and that no other opens can appear in between them. It is suggested that INSERTSTR verbs occur immediately after the OPEN sequence for the dialog or alert they are to be associated with. When the code necessary to implement the specific INSERTSTR sequence is generated, it is inserted in the record structure for that procedure.

This process is repeated for any sequence statements with DITLITEM as an object. The code generated for them is inserted in the corresponding procedure's record. For all but the ITEMHIT blocks, the result of this is a line or two of code. These blocks can contain a

large number of sequence statements of virtually any type and thus can result in numerous code lines being generated. This is code that will be implemented if the Ditl item specified is selected by the user when its dialog or alert is active on the desk top. During this process, any dialog or alert procedures that contain calls to other generated dialog or alert procedures are flagged as being 'not simple'. This means they will be processed with the dependency group procedures.

At this point all procedures have been created and all the sequence statements that will result in code to be included in them have been processed and removed from the stack. The Code Generator now creates temporary text files containing the simple procedures. This is done by processing a procedure shell file for each 'simple' procedure and adding the code stored in the procedure record to it. There are three shell files used and three text files created, one each for alerts, dialogs and user procedures. Only dialog or alert procedures flagged as simple are processed, the rest are added to a separate list of dependency group procedures. These will be processed when the dependency group unit is created.

The next type of sequence statements to be processed are the ITEMHIT = GOAWAY blocks. They are processed sequentially and the code output to a temporary text file. The ITEMHIT = INIT block is the next to be processed. As with the goaway blocks, these sequence statements are processed sequentially but the generated code is stored in a linked list. It is possible for this block to contain no sequence statements and this will simply result in an empty initialization routine. At this point the only thing left in the parse stack is the MENUBAR block.

The MENUBAR block can contain two levels of nested blocks within it. The menu lists specified in the sequence are each represented by a block and nested within them are blocks for each of their menu items that have been specified in the sequence. The application will handle these with nested case statements. The menu items specified have procedures created for them, containing all of the code that is to be implemented when that item is chosen. It is illegal to have any code except menu item blocks inside the menu list blocks.

The menu item procedures are created after the end of their block has been found. All of the code generated for sequence statements enclosed within this menu item block has been stored in a linked list. A shell file, the same as for the user procedures, is used. As each menu list or menu item is processed, its name is used to define a constant whose value is the ID number of the item or list. These constants will make the code easier to read and understand. As the MENUBAR block is processed, a growing text file of menu item procedures is created plus a linked list storing the menu list code created. When the MENUBAR block has been processed, the parse stack should be empty and contain only a header record. This remnant is disposed of and the code generation phase is complete.

The actual process of generating the Pascal code in response to the parse records has been glossed over to this point. It is all handled by a procedure in the Code Generator. This procedure is passed the verb, object, symbol and statement type and returns a linked list of code lines. This list will always contain at least one line of code, and could contain several. The code is generated by defining a series of short strings with parts of the final code lines and concatenating these with information encoded in the object or symbol. Some verbs such as QUIT always generate the same code line. Most code lines require some additional information, usually in the form of parameters to toolbox or SystemCalls procedures. For example opening a generic alert requires the ID number of the alert plus the type of alert it is to be. These values are encoded in the symbol passed and are concatenated with the pre-defined strings defined to make a unique code line.

The code generated may require global variable definitions to be created. Some of these are generic, and some unique. A check is always made to insure that a variable definition does not already exist, to avoid duplicates. If we take the OPEN ALERT example, it would require one generic variable to be defined. Any generic OPEN ALERT will use the same variable. On the other hand each OPEN DIALOG requires the creation of a unique variable. A unique name is created for this variable by combining the dialogs ID number with a

string constant. This same technique is used to insure unique names for dialog or alert procedures.

iii) UNIT CREATION DETAIL

The Code Generator closes the code generation dialog (Figure 17) and displays a dialog informing the programmer that unit creation is now under way.

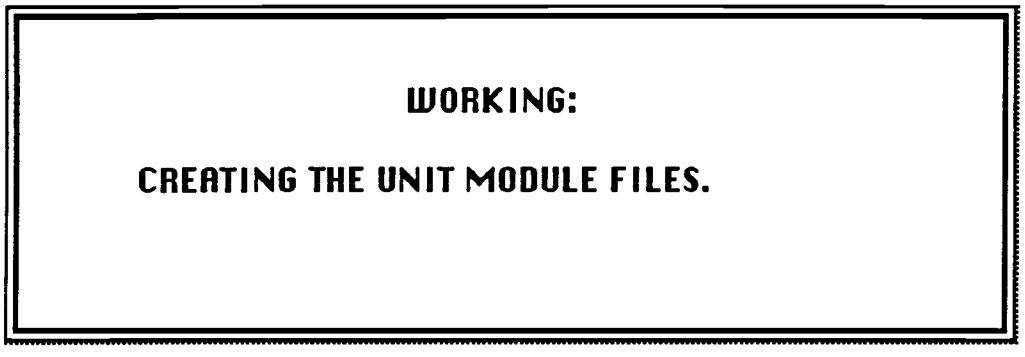

Figure 20. Unit Creation Dialog

Once the parse stack has been processed and all of the necessary code generated, it must be output in the format of a proper LightSpeed Pascal program. This is done by incorporating the generated code stored in the linked lists and temporary text files with shell files to create the units that make up a program. There are at least fourteen unit files output plus a text file listing the build order for these units. There can be more than fourteen output units, but if so the extras are created by repeatedly processing the Dependency Group shell. The shell files have meta symbols indicating where generated code may be inserted. When the shells are processed the meta symbols are removed, but with the exception of the first two in each unit, they do not have to be replaced with code. It is possible to generate a legal program that will run but do nothing from a minimal Sequence Command file.

The unit files can be divided into two groups, the sequence shell files and the system shell files. The latter group is made of four unit shells: OSU_Declarations, OSU_Procedures, OSU_MainEvent and OSU_Main. These represent the core of a Macintosh application. No user specific code is included in them and they have only two

meta symbols. These are replaced with a unique name for each unit and a list of the other units that they are dependant on. These four shell files are not processed by the Code Generator, but rather by the main body of OSU.

The other ten shells are the sequence shell files. The ten sequence shell files all contain the same two meta symbols as the four system shell files, plus up to three more. The exact meaning of the additional meta symbols varies from unit to unit. All of the shell files, including the four system files, are processed in a similar way. Each is read as a text file, and scanned character by character for the meta symbol, '%'. If a character is not a meta symbol, it is written to the output unit file. When a meta symbol is found, the next character is checked to determine which it is. Generated code is then written to the output unit file in place of the meta symbol. If no code has been generated, a comment is added to the output unit in place of the meta symbol.

The first shell file to be processed by the Code Generator is the Dependant Group shell. This is the shell that may produce more than one output unit. Once all of the dialog and alert procedures have been created, those that don't make calls to other dialog or alert procedures are separated from those that do. The latter group will be output as dependency groups. From this list of procedures, all of these that call each other are found and made into one Dependency Group. This may consist of a single procedure, if the only calls it makes are to simple procedures, or it may involve a large group of procedures that make up a call tree. As the dependency group procedures are located and output, they are removed from the list and their procedure records are removed. If after a group has been output, there are still procedures left in the list of dependent procedures, the process is repeated. There is no limit on the number of dependency groups that may be output so this shell could generate a large number of output units.

The reason for these groups was to avoid running into the LightSpeed Pascal limit of 2500 source lines per unit. Once the decision was made to split up the procedures created by the Code Generator, it was necessary to do so in such a way as not to violate

LightSpeed Pascal's hierarchy. All of the Dependant Group units use the same subset of units in an application. See figure 21 below for details.

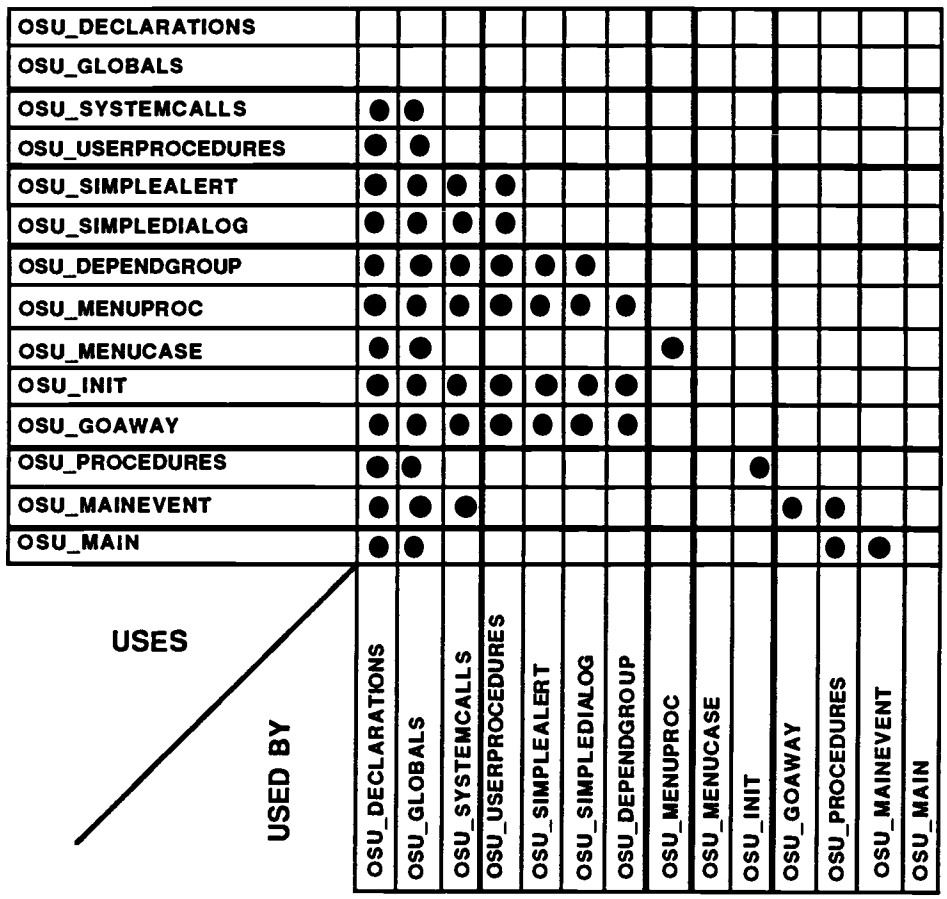

Figure 21. Uses Hierarchy for the Output Units

This figure shows the uses relationships among the output units generated. The dots on the horizontal rows indicate that a unit uses the unit in whose vertical column it occurs. For example the Dependency Group units use: OSU_Declarations, OSU_Globals, OSU_SystemCalls, OSU_UserProcedures, OSU_SimpleAlert and OSU_SimpleDialog. A similar chart showing the uses relationships for OSU is in appendix v.

The next shell file processed is the Init shell. This shell contains the code that is implemented when the application first starts up and is intended to allow the programmer to

generate the initial state for an application. If there is no code, a comment line is added to that effect. A sample of such a unit is shown in Figure 22.

```
{-------------------------------------------------------------------}
{   Copyright 1988, Oregon State University.                        }
{                                                                   }
{ This file is the  UNIT for the Sequence  Initialization code.     }
{ The UNIT OSU_GLOBALS is used, and it uses all of                  }
{ the other generated units that are needed.                        }
{-------------------------------------------------------------------}
UNIT Initialize_username ;

INTERFACE
        USES
                Group1 _UserName, Declarations _UserName,
                Globals _UserName, SystemCalls _UserName,
                UserProcedures _UserName, SimpleAlert _UserName,
                SimpleDialog _UserName ;

        PROCEDURE Seq_Init;

        IMPLEMENTATION

        PROCEDURE Seq_Init; {no parameters}

                BEGIN
                        PROCCALL1;
                        theCursor := GetCursor ( IBEAMCursor );
                        SetCursor (theCursor ^^);
                        Alert303 ( 303 ,screenport) ;
                        Update_Menu(3 , 3 , 1 ) ;
                        Update_Menu(3 , 4 , 1 ) ;
                END;

END. {Initialize _UserName }
```

Figure 22. Sample of a Generated Output Unit

The shells MenuProc and MenuCase are processed next. These will contain the code generated for the Menubar block in the Sequence Command file. MenuCase includes the code for menu lists and MenuProc, the procedures generated for the menu items. If no code is added to the MenuCase unit this would also mean that no procedures have been added to the MenuProc unit. It is possible for there to be code added to the MenuCase unit but still have none added to the MenuProc unit.

The name of the unit in Figure 22 was generated, as was the list of names in its USES clause. This was done by replacing the 'OSU_' string with a string entered by the user,

'UserName' in this case. The lines of code inside Procedure Seq_Init were generated in response to sequence statements in the initialization block of the Sequence Command file. If that block had been empty, a comment would have been included in place of the code lines. Finally the name of the unit was added to the comment at the end of the unit.

The next shell processed is the Globals shell. Any constants or variables that have been created as a result of code generation are added to it. The last five shell files processed by the Code Generator are: SystemCalls, UserProcedures, SimpleAlerts, SimpleDialogs, and GoAway. SystemCalls contains only the two meta symbols that the system shell files have. The UserProcedures unit has those procedure stubs created for the DO verb sequence statements added. Such a procedure was shown in figure 19. The two units SimpleAlerts and SimpleDialogs have those procedures generated for alerts or dialogs, that are not included in the Dependent Group units, added. The GoAway shell has the code added for actions to be taken when window goaway boxes are selected by the user.

In actual fact the Code Generator does not first generate all of the code and then output all of the units. In some cases, such as the Initialization code block, the output unit is created as soon as that section of the parse stack has been processed. This was to allow all of the processing of some types of sequence statement to be concentrated in one place. Conceptually it is easier to follow if the processing is explained as if it were sequential functions. The Initialization and Dependency Group units are output before the code generation is completely finished. The MenuCase and MenuProc units are output as soon as the parse stack has been processed. Then the rest of the units are created in the order described.

6) RELATIONSHIP TO OTHER SECTIONS OF OSU

i) OSU MAIN BODY

The main body of OSU handles the processing of the four system shells: OSU_Declarations, OSU_Procedures, OSU_MainEvent and OSU_Main. These are done in the same way as the Code Generator's processing of its shells and indeed makes use of some of the same utility functions. It is also possible to enter the Code Generator without first going through the Graphical Sequencer. In this case the user is prompted for the names of the Sequence Control and Sequence Command files to be used.

The main body collects one vital input needed for the processing of all the shell files. This is a string called 'tempfn' that the user enters in a dialog. It is combined with parts of the shell names to create unique names for the output units. It is also used to create unique names for temporary files created during processing. The only other global variable needed is a disk volume used to insure that all files created, both temporary and unit, are in the same place.

ii) RESOURCE EDITOR

There is no direct interface between the Code Generator and the Resource Editor, RezDez. If the entire sequence of OSU sections is followed, the Graphical Sequencer occurs between RezDez and the Code Generator in the control flow. The only connection is indirect. There must be resources on which the Sequence Command and Sequence Control files are based and these may be created in RezDez, but they could be created with another tool or be obtained from a pre-existing resource file. [Bose 88]

iii) GRAPHICAL SEQUENCER

The Graphical Sequencer has a direct interface with the Code Generator. In fact the output of this section of OSU, the Sequence Command file and the Control List file, are the two inputs to the Code Generator. These can be created or modified by the programmer outside

the Graphical Sequencer, but it is intended that they usually originate within the Graphical Sequencer.

These two sections of OSU are really two halves of a whole. The intent was to allow the programmer to specify a sequence of events that occur in response to user actions within an application. The Graphical Sequencer allows the programmer to specify such a sequence for the objects in any resource file. The Code Generator then creates the source code for a complete application to implement this sequence. Without the sequence, the Code Generator cannot create an application, and without the generated application, the Graphical Sequencer cannot produce any real output for the programmer. [Handloser 88]

iv) PLUM DIAGRAMMER

This part of OSU is still under design so it is not possible to state with any authority what it will do or how it will relate to the Code Generator. This section is intended to allow the automatic generation of source code routines derived from Plum Diagrams. The purpose of this is to produce the code that is unique to a specific application and cannot be created by the Code Generator. This code may replace the procedure stubs created from the DO verb sequence statements. This would allow full-fledged application prototypes to be generated automatically.

Details of how this will be done are yet to be worked out. Possibilities include processing the UserProcedures unit produced by the Code Generator and filling in the stubs generated or modifying the Code Generator so it will insert the code generated in the Plum Diagrammer. The latter of these will have to be used if parameters are to be added to the procedures themselves and to calls generated from the DO verb sequence statements. Further details will have to wait until this project is completed. [Hsieh 88]

7) SUMMARY AND CONCLUSIONS

i) CONCLUSION

The problem this program set out to solve was the automatic generation of compilable source code to form working prototype applications on the Macintosh. This was accomplished. The programmer is allowed to specify a sequence of actions based on user interaction with the objects in the resource file. The Sequence Command Language was developed to encode this sequence. This sequence, plus the shell file templates of an application, serve as inputs to the Code Generator and the source code units of a complete application are the output. These units can be compiled and linked with a binary Resource file to form a working prototype.

This tool has already been used by students for the development of applications and has great potential for use by Macintosh programmers. Work continues on enhancements to the existing phases of OSU and on the addition of new phases.

ii) LIMITATIONS

The Code Generator can only generate code for those actions that can be specified with the Sequence Command Language. This was designed to support the most common actions possible of most of the pre-defined resource types allowed in a resource file. It is by no means an exhaustive list , and it is quite possible a programmer will want to use actions or resource types not supported. The code is generated only in Pascal, and only in a format supported by Think Technologies LightSpeed Pascal compiler. If it was to be used with another compiler, the code would have to be re-formatted by the programmer.

The code currently generated for ditl itemhits is very limited. In a typical Macintosh application the control or ditl items are handled in two separate blocks of code. First a block that handles any changes in the appearance of controls and is exited when the user selects one specified button item. Second a conditional block that take any delayed action signaled

by the state of the controls when the first block is exited. Presently all of the code generated is contained in the first block section. The framework of the second is generated, but it contains no user specific code. Also the Code Generator does not automatically take care of updating specific control types, even though this information is now encoded in the Sequence Command language. For example checking and un-checking check boxes as they are selected. This must be encoded in the sequence by the programmer now but with the information encoded in the sequence, it could be handled automatically by the Code Generator.

It is common practise in Macintosh programming to use the same dialog from the resource file for a number of different purposes. Static text messages can be changed as needed using paramtext. Also the user input via controls can be interpreted differently to suit different implementations. The Code Generator currently does not allow the generation of different procedures that use the same resource object. Yet this is a simple matter if the source code is being created manually. A complete solution to this problem was not found during the development phase and it remains to be solved in future enhancements to OSU.

The generation of procedures to handle user specific windows is not possible, indeed windows cannot even be specified in the sequences created in the Graphical Sequencer. This is because windows are not implemented as generically as dialogs by the routines in the toolbox. Another limitation is the requirement that INSERTSTR verbs must be physically located after the open sequence for the alert or dialog they are to be associated with. This is an undue restriction of the format of the Sequence Command file and could be avoided if more information were encoded in the symbol of the INSERTSTR sequence. Also at present only one string can be inserted into a dialog even though Macintosh provides for up to four. This is a limitation of the way the code for INSERTSTR sequences is generated.

iii) RECOMMENDATIONS FOR FUTURE ENHANCEMENTS

1) CHANGES TO THE ITEMHIT = DITLITEM SEQUENCES

ITEMHIT = DITLCASE [ID#=type code]
ITEMHIT = DITLITEM [ID#=type code]

The first would go into the block for that control in the procedure's case statement and should include immediate actions such as radio buttons being turned on and off. The second would cause the code to be inserted in the conditional block for that control number. These would be code actions that occur when the immediate action block is exited, such as the result of choosing a particular radio button.

The actual code to handle the appearance of controls such as radio buttons would still have to be specified at the Graphical Sequencer level. Information such as groupings of radio buttons and the types of all ditl items associated with procedure are not available at the code generation level, and obtaining them would require access to the resource file from the Code Generator. However the Code Generator could handle the changes in appearance of check boxes and push buttons automatically, if any sequence using that ditl item was in the Sequence Command file. The control type is encoded in the symbol for any ditl item used, but if a ditl is not specified, the Code Generator has no way of knowing what it is and thus what code to generate for that item.

2) ADDING SECOND STAGE META SYMBOLS

It would be relatively easy to give the programmer the option to set a flag that would cause meta symbols to be added to the output units in the form of comments. This would allow the output units to be further processed by a later stage of the program, while not impeding the use of them in a prototype project. These comments would be of the format {%symbol} where symbol is a predefined value. These could be inserted in the procedure shells for procedures generated by the do verb and be replaced by code generated in the Plum Diagrammer. A similar meta symbol could also be inserted to be replaced with the declaration block for these procedures.

This change would require adding another dialog to the user interface to allow the programmer to choose to add these symbols. Then when creating the code the appropriate meta symbol comment would be inserted. Similarly when the do verb procedures are created, meta comments could be inserted if the flag was set. Of course the routines to process the output units and generate and insert code in place of the second stage meta symbols would also have to be written.

3) CHANGES TO THE DO VERB SEQUENCES

It would be possible to add some parameters to the DO procedure calls either by adding an object, it does not have one now, or by adding more information to the symbol. It currently contains the name of the procedure only. One example:

DO CALL [name=parameters]
DO DEFINE [name=parameters]

The first would simply generate a call to a procedure called 'name' with the parameters in the string in the braces. This would be a list of one or more variable names separated by commas. It could be textually included in the call statement without any further processing. The second would cause a procedure shell to be generated with the name in the string passed and the parameter string in this case would be the formal parameter definitions with type. Again this string could be inserted as a unit without any processing. This method would allow calls to be generated, with parameters, for procedures generated by the DO DEFINE grouping or elsewhere such as the Plum Diagrammer. It would also still allow the programmer to define procedure stubs. The variables used as parameters in the DO CALL statements would have to be formally defined at some level.

4) ADDING DECLARATION DEFINITION:

There is at present no direct way to define local or global constants, variables or types. Some are generated as a side effect of code generation but the programmer has little control over these. If a new type of statement was added, perhaps these could be supported.

DEFINE [type=definition]

The type would be constant, type, or var. The definition would be a string that represented the actual definition statement, for example [var=done : boolean].
There is support at the present time for local constant and variable definitions in the dialog and alert procedures generated, and for global ones as well. Types could easily added with little change to existing code. This sequence could be used to make both local or global declarations.

5) CHANGES TO THE INSERTSR SEQUENCES

At the present time the INSERTSTR verbs are associated with a particular dialog or alert by physical proximity in the Sequence Command file. The INSERTSTR is presumed to go with the first OPEN DIALOG or OPEN ALERT above it in the parse stack. This information could be encoded in the symbol for the INSERTSTR sequences to avoid this restriction.

INSERTSTR 4 [101=D202] ;

The number 4 would still represent the place in the paramtext toolbox call and the number 101 would still be the ID number of the string from the resource file that was to be inserted. The addition is the '=D202' section of the symbol. The letter would be an 'A' for alert or 'D' for dialog. The number would be the ID number of the dialog or alert. This would remove the last sequence statement that enforces the relative positioning of related statements in order to communicate information.

6) ADDITION OF WINDOW PROCEDURES:

The Code Generator presently does not support the generation of procedures to handle windows. Windows can be opened or closed using the routines in the SystemCalls unit but that is all. This feature was originally intended to be supported, but during development of both the Graphical Sequencer and the Code Generator, no approach was found to handle windows the way dialogs are. A lot of work would be needed to support this. Modifications would need to be made to both the Graphical Sequencer and the Code Generator to allow window procedures to be specified and created automatically. There are

also procedures needed to support the window control types. In the SystemCalls unit there are stubs that would need to be completed to support the re-sizing and dragging of windows. Also routines should be developed to support the use of scroll bars. The goaway box is supported presently by the Code Generator.

These enhancements are not essential to make the Code Generator a useful tool. They would allow it to take even more of the burden of programming the user interface of a Macintosh application off the programmer. The more features commonly used in Macintosh applications that can be automated within OSU, the closer it will come to automatically producing complete applications for the programmer.

8) Bibliography

1) Apple Computer, **Inside Macintosh Promotional Edition,** Apple Computer, Inc. 1985.

2) Apple Compute, **Inside Macintosh Volume IV,** Addison Weseley Publishing Inc. 1986.

3) Bergmark, Donna , "Gibsgen: Code Generation for Gibbs", **Proceedings: Second Conference On Software Development Tools, Techniques, and Alternatives,** Computer Science Press, 1985.

4) Boehm, Barry, "Software Engineering", IEEE **Transactions on Computers,** December 1976.

5) Bose, Sharada, **RezDez: A Tool for Specifying Graphical Objects in OSU,** Master's Research Paper, Oregon State University, 1988.

6) Friman, Bertil, "MGEN - A Generator For Menu Driven Programs", IEEE **Transactions on Software Engineering,** 1984.

7) Friman, Bertil, "X - A Tool For Prototyping Through Examples",**Proceedings: Second Conference On Software Development Tools, Techniques, and Alternatives,** Computer Science Press, 1985.

8) Glickman, Stuart, and Becker, Mark, "A Methodology for Evaluating Software Tools", **Proceedings: Conference On Software Tools,** Computer Science Press, 1985.

9) Handloser, Fredrick, **A Graphical Rapid Prototyper for the Macintosh** ,Master's Research Paper, Oregon State University, 1988.

10) Hsieh, Chia-chi, **A Graphical Editor for Pascal Programming on Macintosh,** Master's Research Paper, Oregon State University, 1988.

11) Lee, Chin Kwan, **Macgen**, Master's Research Paper, Oregon State University, 1986.

12) Lewis, T. G., "Apple Macintosh Software", IEEE **Software**, March 1985.

13) Lewis, T. G., and Yang, Sherry, and Handloser, Fredrick, and Bose, Sharada, **Prototypes From Standard User Interface Management Systems,** Computer Science Department, Oregon State University, 1988.

14) Lightspeed Pascal Software, **Lightspeed Pascal User's Guide and Reference Manual,** THINK Technologies, Inc. , 1986

15) Luqi, and Ketabchi, Mohammad, **A Computer Aided Prototyping System,** Naval Postgraduate School, 1987

16) Madhavji, Nazim, "Fragtypes: A Basis for Programming Environments", IEEE **Transactions on Software Engineering,** January, 1988.

17) Meyers, B.A., **Creating User Interfaces by Demonstration, PhD Dissertation,** Department of Computer Science, University of Toronto, May 1987.

18) Meyers, B.A., "Tools fro Creating User Interfaces: AN Introduction and Survey, IEEE **Software,** January 1989.

19) Schulz, Arno, "CAS A Tool For The Interactive Program Design", **Proceedings: Conference On Software Tools,** Computer Science Press, 1985.

20) Takatsuka, Jim, and Huxham, Fred, and Burnard, David, **Using the Macintosh Toolbox with C,** Sybex, 1986.

21) Van Hoeve, Frans and Engmann, Rolf, "An Object-oriented Approach to Application Generation", **Software - Practise and Experience** , September, 1987.

22) Yang, Sherry, and Lewis, T. G., and Hsieh, Chia-Chi, **OSU: Integrating CASE and UIMS,** Computer Science Department, Oregon State University, 1988.

APPENDICES

9) APPENDICES

i) DEFINITIONS

1) THE PROGRAM ENVIRONMENT

Oregon SpeedCode Universe (OSU): a Macintosh program designed to function as a programmers development workbench. It is made up of several Master's projects developed by computer science students at Oregon State University. It is an integrated application development environment designed to increase programmer productivity by providing rapid prototyping, reusable units, program generation and expert systems technology. [Yang 88]

Resource Editor (Rez Dez): A tool to allow a programmer to graphically create and design all of an application's resources, one phase of OSU [Bose 88].

Graphical Sequencer: A tool to allow the programmer to manipulate the screen objects graphically to specify the sequence of a user interface, one phase of OSU [Handloser 88].

Code Generator: A tool to generate source code based on a user interface sequence encoded in a data file, one phase of OSU.

Sequence Command file: the data file created by the Graphical Sequencer which encodes a sequence of user interactions with a set of objects in a resource file.

Sequence Command Language: the syntax developed for OSU to encode the sequence of user actions.

Control List file : The Graphical Sequencer provides this as a second input to the Code Generator. A file containing a list of the ID numbers and types of all of the windows, dialogs, and alerts contained in a resource file. The file also contains a count of the number of items in the dialog item list

for each object. This information is needed by the Code Generator to create the output application.

shell file : A text file that contains a template for a unit or procedure of Pascal source code. These are used in code generation.

meta symbol : Special symbols contained in shell files that are replaced with generated code. They consist of the per cent character followed by a digit. EG %1, %2

Dependency Group: A set of generated procedures that are dependent on each other. They make calls to other procedures in the group or to other generated procedures. The group may contain one or more procedures.

2) MACINTOSH PROGRAMMING TERMS

User Interface Toolbox (**toolbox**): a level above the operating system that provides support to implement the standard Macintosh user interface in an application. The toolbox calls the operating system to do low-level operations and both levels can be called directly from an application. [Apple Computer 85]

SFGETFILE, **SFPUTFILE**. Two toolbox routines that are used in the sequence command language. These routines allow the user to look for a file to open for reading, SFGETFILE, and specify a file to use for writing, SFPUTFILE. [Apple Computer 85]

application : a finished program. It consists of two compiled input files that have been linked: the object code file and the binary resource file. {Lewis 85]

Main Event Loop : A loop inside the source code of a Macintosh application that handles the input from the keyboard and mouse. It detects which objects the mouse is on when the button is pushed and directs this input to the appropriate place. [Takatsuka 86]

resource file : a data file that stores the definitions of the resources used by a Macintosh application. also known as the Resource fork. [Takatsuka 86]

Resources : the graphical objects used by the Macintosh to make the visual interface. There are many types of pre-defined resources such as menus, windows, dialogs, and cursor types. A user can also define new ones. Toolbox and Operating System routines exist to support use of the pre-defined resources. [Apple Computer 85]

3) MACINTOSH RESOURCE TYPES

desktop: a metaphor for the Macintosh user interface. {Lewis 85]

menus: Objects that represent the commands available to a user on the desktop. They allow the user of an application to examine all choices available to them at any time without having to choose one or to remember command words or keys. [Apple Computer 85]

menu bar :the white strip across the top of the Macintosh screen. It contains words or symbols, each of which specifies a title. [Apple Computer 85]

menu lists : the titles in the menu bar. [Apple Computer 85]

menu items : the items in a menu list.[Apple Computer 85]

window : an object that appears on the desktop and presents information to the user. They are usually rectangular or square and may contain several pre-defined controls inside their boundaries. Windows are most commonly used to manipulate text or graphics. [Apple Computer 85]

goaway box: a small box in the upper left hand corner of a window, it closes the window if it is selected. [Apple Computer 85]

scroll bar along the bottom, and possibly also the right hand edges, it is used to change that portion of a window's data which is displayed. [Apple Computer 85]

frontmost : the active window, indicated with a highlighted title bar. There can be a number of windows on the screen at any one time, but only

one can be active. Also the controls of any inactive windows will be invisible. [Apple Computer 85]

dialog : a box that appears on the screen when the application needs more information to carry out a command. Dialogs can display messages, contain controls to collect responses and provide rectangles for the user to enter text in. [Apple Computer 85]

modal dialog : a dialog that requires the user to respond before they can do anything else. [Takatsuka 86]

modeless dialog : a dialog that does not require the user to respond before doing anything else.[Takatsuka 86]

alert : a type of dialog box. It is used when something has gone wrong or something must be brought to the user's attention. There are four types. Alerts can have sound associated with them. One or more beeps from the Macintosh's speaker typically accompanies the alert. Alerts are essentially a specialized form of modal dialog. [Takatsuka 86]

plain, **caution**, **stop**, **note**. : types of alerts. The last three all have a small icon that appears in the upper left corner. [Takatsuka 86]

control : an object on the Macintosh screen with which the user, using the mouse, can cause instant action or change settings to modify a future action. There are a number of pre-defined control types and they can be displayed inside a dialog or an alert. [Apple Computer 85]

button :a control that appears as a rounded corner rectangle with a title inside. They cause an immediate or continuous action when selected. [Apple Computer 85]

check box : a control that appears as a small square with a title beside it. Check boxes retain and display a setting, selecting it changes the setting from displaying an x to empty or vica versa. [Apple Computer 85]

radio button : a round check box and they too retain and display settings, in this case a black or white circle. Radio buttons are grouped and only one member of a set may be selected at a time. Selecting another causes the first to be un-selected. [Apple Computer 85]

Static text boxes are rectangles that display text the user cannot change in any way. Static text boxes can contain special symbols. [Apple Computer 85]

paramtext: special symbols ('^0', '^1', '^2', or '^3') in a static text box that can have text substituted for them.The user still cannot alter the text in any way, but this allows the same static text box to display a variety of preset messages to the user. [Apple Computer 85]

ditl item : term used in OSU to refer to the variety of controls that can be present in an alert or dialog. This is short for 'dialog item list' item.

ditl list : a data structure attached to a dialog that contains information on the controls associated with that dialog. [Apple Computer 85]

icon : a 32 bit by 32 bit graphical image that graphically represents an object, concept or message. Typically all applications are represented on the screen as icons. Also files, disks and the trash can are icons. Icons can also be used in the place of menu items or controls. [Apple Computer 85]

picture : a transcript of the Macintosh calls needed to reproduce a graphical image. It is also used to refer to the image itself in OSU. A picture is drawn inside a frame and the toolbox routines that draw the picture automatically scale it to fit the frame created for it. [Apple Computer 85]

rectangle : the frame created to contain a picture. [Apple Computer 85]

cursor : a 16 bit by 16 bit image that moves around the screen in response to mouse movements by the user. The user can design any shape they wish for the cursor but there is a set of pre-defined images. [Apple Computer 85]

arrow, **cross**, **plus**, **watch**, **ibeam** : pre-defined cursor images. [Apple Computer 85]

select : user choice of an objects by positioning the cursor over it and pressing the mouse button.

itemhit : term used within OSU to refer to the user's selection of an object. This could be done with either the mouse or the keyboard.

4) PROGRAM LANGUAGE AND DEVELOPMENT SYSTEM

project : a program to be compiled with the LightSpeed Pascal Compiler. The source code divides into units of not more than twenty-five hundred lines of source code. The collection of units is called a project. [LightSpeed 86]

units : The independently compiled modules a project is divided into. They contain an interface section, an implementation section and have 'END.' at the end. [LightSpeed 86]

interface : a section of a unit containing definitions of the data types, constants and variables, and of the procedures and functions from a unit that may be available to other units. [LightSpeed 86]

implementation : a section of a unit containing the actual source code of all procedures and functions in the unit. It also includes the definitions of any data types, constants and variables that are hidden from other units. [LightSpeed 86]

uses clause : a line contained in the interface section of a unit that allows it to list the names of other units whose interface sections it wishes to access. A unit may not use

another unit unless it has been previously defined in the project's list of units. Thus there is a strict hierarchy enforced as to which units can use which others. [LightSpeed 86]

ii) SEQUENCING LANGUAGE GRAMMAR

Grammar for the Command Language of the Code Generator

<SEQUENCE> ::= <CONDITION1>{<CONDITION3>{<CLAUSE1> }* . ; | {<CLAUSE1>}*}
<CONDITON2> { <CONDITION4> { <CLAUSE2> }* . ; | { <CLAUSE1> }*

<CONDITION1> ::= ITEMHIT = INIT
<CONDITON2> ::= ITEMHIT = MENUBAR
<CONDITION3> ::= <VERB1> = <SPACE> <OBJECT1>
<CONDITION4> ::= <VERB1> = <SPACE> <OBJECT11>

<CLAUSE1> ::= <CONDITION3> | <ACTION> | <COMMENT>
<CLAUSE2> ::= <CONDITION4> | <ACTION> | <COMMENT>

<ACTION> ::= <VERB2> <SPACE> <OBJECT2> | <VERB3> <SPACE> <OBJECT3> |
<VERB4> <SPACE> <OBJECT4> | <VERB5> <SPACE> OBJECT5> |
<VERB6> <SPACE> <OBJECT6> | <VERB7> <SPACE> <OBJECT7> |
<VERB8> <SPACE> <OBJECT8> | <VERB9> <<SPACE> OBJECT9> |
<VERB10> <<SPACE> OBJECT10> | <VERB11> |
<VERB12> <SPACE> <OBJECT12>

<COMMENT> ::= *{STRING}

<VERB1> ::= ITEMHIT =
<VERB2> ::= OPEN
<VERB3> ::= FRONT | BACK
<VERB4> ::= ENABLE | DISABLE
<VERB5> ::= PUTIN
<VERB6> ::= CHANGE
<VERB7> ::= INSERTSTR
<VERB8> ::= CLOSE
<VERB9> ::= DO
<VERB10> ::= TAKEOUT | ADD
<VERB11> ::= QUIT
<VERB12> ::= CHECK | UNCHECK

<OBJECT1> ::= DITILITEM <DITLID> | GOAWAY <RESID>
<OBJECT2> ::= WINDOW <RESID> | DIALOG <RESID> |
DIALOG <PRESID>| ALERT <ALERTID>
<OBJECT3> ::= WINDOW <RESID> | DIALOG <RESID> | DIALOG <PRESID>
<OBJECT4>::= <USERDEFINED> | DIALOG <RESID>
<OBJECT5>::= <ICONSTR>
<OBJECT6> ::= CURSOR <STATE>
<OBJECT7>::= <PLACE> <RESID>
<OBJECT8> ::= WINDOW <RESID> | DIALOG <RESID>
<OBJECT9>::= <USERPROCEDURE>
<OBJECT10> ::= <PICTURE> <PICID>
<OBJECT11> ::= <USERDEFINED> | DIALOG <RESID>
<OBJECT12> ::= <MENUITEM> | DITILITEM <CTLID>
<OBJECT13> ::= DIALOG <RESID>| ALERT <ALERTID>

<PRESID> ::= [SFGETFILE | SFPUTFILE]
<STATE> ::= [ARROW | WATCH | IBEAM | CROSS | SHOW | HIDE | OBSCURE]
<RESID> ::= [<INTEGER>]
<PICID> ::= [<INTEGER>=<INTEGER>=<INTEGER>=<INTEGER>=<INTEGER>]
<ALERTID> ::= [<INTEGER>=<ATYPE>]
<DITLID> ::= [<INTEGER>=<PTYPE><RESOURCEID>=<CTYPE>]
<CTLID> ::= [<INTEGER>=<PTYPE><RESOURCEID>=<INTEGER>]
<USERDEFINED> ::= MENULIST | MENUITEM
<ICONSTR> ::= [(D|A|M)=<RESOURCEID>=<ITEMID>]
<MENULIST> ::= [<RESOURCEID>=0]
<MENUITEM> ::= [<RESOURCEID>=<ITEMID>]
<RESOURCEID> ::= <INTEGER>
<USERPROCEDURE> ::= string
<ITEMID> ::= <INTEGER>
<PLACE> ::= 1 | 2 | 3 | 4
<ATYPE> ::= 1 | 2 | 3 | 4
<PTYPE> ::= A | D
<CTYPE> ::= U | B | C | R | S | E | I | P
<SPACE> ::= ' ' { NOTE: THIS INDICATES A SINGLE SPACE CHARACTER }

GENERAL NOTES

1) **All** lines will end with a : space ;<CR>

2) Each level of nesting in the sequence will be indented three spaces.

3) An empty sequence command file will still contain a CONDITION1 and a CONDITION2 (see example next page)

4) Comment Lines must have an * as the first character and the rest of the line is then ignored. Blank lines are similarly ignored entirely.

iii) USAGE DETAILS

1) ALERTS: OPEN ALERT [RESOURCEID=TYPE] ;

ResourceID is the Alert's ID number in the resource file and type determines whether it is a STOPALERT(1), NOTEALERT(2), CAUTIONALERT(3), or by default for any other value, a regular ALERT. If the ALERT is a procedure then OPEN is a call to it, if not it is a call to a low level ALERT handling routine in the System Calls shell file.

EG: OPEN ALERT [223=2] ;

OPEN ALERT [233=1] ;

2) CURSOR:

CHANGE CURSOR [ARROW] ; { causes an InitCursor() call}

CHANGE CURSOR [PLUS] ; { next 4 set the cursor to the }

CHANGE CURSOR [WATCH] ; { selected predefined type}

CHANGE CURSOR [IBEAM] ;

CHANGE CURSOR CROSS] ;

CHANGE CURSOR [SHOW] ; {causes a ShowCursor call }

CHANGE CURSOR [HIDE] ; {causes a HideCursor call }

CHANGE CURSOR [OBSCURE] ; {causes an ObscureCursor call}

3) DIALOG:

OPEN DIALOG [resourceID] ;

OPEN DIALOG [SFGETFILE] ;

OPEN DIALOG [SFPUTFILE] ;

CLOSE DIALOG [resourceID] ;

CLOSE DIALOG [SFGETFILE] ;

CLOSE DIALOG [SFPUTFILE] ;

FRONT DIALOG [resourceID] ;

BACK DIALOG [resourceID] ;

In the case of the OPEN DIALOG [resourceID] these will generate a call to a procedure with the name 'DidresourceID' if it has been created. This is caused when an Insertstr or Itemhit = DitlItem finds an OPEN dialog as the first open above it when parsing. Otherwise it is a call to low level dialog handling routines in a shell file. If a Dialog is a procedure then a close dialog **inside** a DitlItem Itemhit will generate an exit to that procedures itemhit loop, and then to the procedure. Otherwise it will be treated as a non-fatal Parse error to attempt to close a dialog that is a procedure.

If not it results in a call to low level dialog handling routines in a shell file.

NOTE: SFGETFILE and SFPUTFILE opens and closes will never generate procedures.

RE: FRONT and BACK, although I generate code for these, it is my understanding that their use is on hold at the present time.

EG:

OPEN DIALOG [324] ;

OPEN DIALOG [SFGETFILE] ;

OPEN DIALOG [SFPUTFILE] ;

CLOSE DIALOG [324] ;

CLOSE DIALOG [SFGETFILE] ;

CLOSE DIALOG [SFPUTFILE] ;

FRONT DIALOG [325] ;

BACK DIALOG [325] ;

4) DITLITEM:

ENABLE DITLITEM [DitlID] ;

```
{ results in a call to HiliteControl with a value of 0 }
DISABLE DITLITEM [DitlID] ;
{ results in a call to HiliteControl with a value of 254 }
CHECK DITLITEM [DitlID] ;
{ results in a call to SetCtlValue with a value of 0 }
UNCHECK DITLITEM [DitlID] ;
{ results in a call to SetCtlValue with a value of 1 }
ITEMHIT = DITLITEM [CtlID] ;
```

NOTE: all statements between this and the next . ; statement which result in code generation are included in the case item for this control in the appropriate procedure . DitlID is of the form [CtlID=A/DResID=Ctl#].

The DitlID is the control's number in the dialog/alert Ditl list. The A/D indicates whether this goes in a dialog procedure or an alert procedure, and the ResID is the ID number of that dialog or alert. The Ctl number determines which control's case this code is to be inserted in , in the procedure being generated. The CtlID has the same format except the last item indicates the control type rather than a number. These types are singles characters representing the first letter of the controls name.

U: userdefined, B: Button, R: Radio, E: Edit text, S:static text, I : Icon, C: check box, P: Push button.

EG:

```
ENABLE DITLITEM [3=D212=2] ;
DISABLE DITLITEM [2=D212=2] ;
CHECK DITLITEM [4=A224=2] ;
UNCHECK DITLITEM [4=A224=2] ;
ITEMHIT = DITLITEM [1=D212=R] ;
ENABLE DITLITEM [1=D212=1] ;
```

DISABLE DITLITEM [2=D212=1] ;

DISABLE DITLITEM [3=D212=1] ;

DISABLE DITLITEM [4=D212=1] ;

5) ICON: PUTIN IconID [A=resourceID=itemID]

PUTIN IconID [D=resourceID=itemID]

PUTIN IconID [M=MenuListID=MenuitemID]

NOTE: Although code is generated for these, it is my understanding that their use in OSU is on hold at the present time.

EG:

PUTIN 234 [A=224=3] ;

PUTIN 234 [D=212=4] ;

PUTIN 234 [M=3=2] ;

6) INIT: ITEMHIT = INIT ;

NOTE: all statements between this and the next . ; statement which result in code generation are included in the Seq_Init procedure . It can be empty in which case that procedure has no effect.

EG:

ITEMHIT = INIT ;

OPEN DIALOG[212] ;

ENABLE FILE [2=0] ;

DISABLE EDIT [3=0] ;

. ;

7) MENUBAR ITEMHIT = MENUBAR ;

NOTE: all statements between this and the next . ; statement which result in code generation are included in the Menu_List case statement or the procedures it calls. It can be empty in which case there is no effect.

EG:

```
ITEMHIT = MENUBAR ;
 ITEMHIT = EDIT [3=0] ;
  ITEMHIT = CUT [3=1] ;
   DO [SORT] ;
   DISABLE COPY [3=2] ;
   DISABLE PASTE [3=3] ;
   DISABLE CLEAR [3=4] ;
  . ;
  ITEMHIT = COPY [3=2] ;
   QUIT ;
  . ;
. ;
 ITEMHIT = SEARCH [4=0] ;
  ITEMHIT = FIND [4=1] ;
   DO [SORT] ;
   ENABLE COPY [3=2] ;
   ENABLE PASTE [3=3] ;
   ENABLE CLEAR [3=4] ;
  . ;
  ITEMHIT = CHANGE [4=2] ;
   OPEN ALERT [224] ;
  . ;
. ;
```

. ;

8) MENU

A: MENULIST:

ENABLE MENUITEM [MenuList=0] ;

{ results in a call to EnableItem }

DISABLE MENUITEM [MenuList=0] ;

{ results in a call to DisableItem }

ITEMHIT = MENUITEM [MenuList=0] ;

NOTE: all itemhits on menu items between this and the next . ; statement are included as cases in the case statement generated for this MenuList. The name for this procedure is the string in MENULIST with DO prefixed to it. Also a constant with the name of the string in MENUITEM with an underscore appended and a value of MenuList. EG ITEMHIT = FILE [4=0] results in a constant FILE_ with a value of 4 and a procedure DOFILE. This procedure is called by the case statement in the Menu shell with the case index being the constants defined. Any code generated for menu items whose MenuList has the same value will be included in the procedure. It can be empty in which case that case has no effect.

EG:

ENABLE FILE [2=0] ;

DISABLE EDIT [3=0] ;

ITEMHIT = SEARCH [4=0] ; {See B. below}

B: MENUITEM:

ENABLE MENUITEM [MenuList=itemID] ;

{ results in a call to EnableItem }

DISABLE MENUITEM [MenuList=itemID] ;

```
{ results in a call to DisableItem }
CHECK MENUITEM [MenuList=itemID] ;
{ results in a call to CheckItem with a value of true }
UNCHECK MENUITEM [MenuList=itemID] ;
{ results in a call to CheckItem with a value of false }
ITEMHIT =  MENUITEM [MenuList=itemID] ;
```

NOTE: all statements between this and the next . ; statement which result in code generation are included in the procedure generated for this menu item. The index for this case is a constant with the name of the string in MENUITEM with an underscore appended and a value of ItemID. EG ITEMHIT = OPEN [3=2] results in a constant OPEN_ with a value of 2. This case is included in the MenuList procedure whose constant definition has a value of 3. It can be empty in which case that case has no effect.

EG:

```
ENABLE CLOSE [2=3] ;
DISABLE DELETE [2=6] ;
UNCHECK COPY [3=2] ;
CHECK CLEAR [3=4] ;
ITEMHIT =  EDIT [3=0] ;
  ITEMHIT =  CUT [3=1] ;
    DO  [SORT] ;
    DISABLE COPY [3=2] ;
    DISABLE PASTE [3=3] ;
    DISABLE CLEAR [3=4] ;
  . ;
  ITEMHIT =  COPY [3=2] ;
    QUIT ;
```

. ;

. ;

9) PICTURE

ADD PICTURE [resourceID=x1=y1=x2=y2] ;

TAKEOUT PICTURE [resourceID] ;

The ADD results in picture resourceID being drawn inside a rectangle that has been created with an upper left corner of (x1,y1) and a lower right of (x2,y2). TAKEOUT causes that picture to be erased and the rectangle re-drawn with corners of (0,0) and (0,0).

NOTE: Although I generate code for these, it is my understanding that their use is on hold at the present time.

EG:

ADD PICTURE [115=20=20=100=200] ;

TAKEOUT PICTURE [115] ;

10) QUIT: QUIT ;

This is used to exit the shell program and generates the code line "didWeQuit := true;". This value is used by the MainEventLoop and DoCommand procedure to flag an exit.

11) STRING: INSERTSTR PLACE [stringID]

This will result in a Paramtext call being generated and placed in a procedure for the first open dialog or alert detected above the Insertstr call. If this procedure does not exist, this will cause it to be generated. If there is no such open it is a syntax error. The PLACE value is a 1..4 and indicates which parameter of the call the string should be. The ID is the string's resource ID.

EG:

INSERTSTR 1 [16] ;

INSERTSTR 2 [17] ;

12) USERPROCEDURE: DO [name]

This will cause an empty procedure called name to be generated if it does not already exist and always result in a call to procedure name.

EG: DO [SORT] ;

13) WINDOW:

OPEN WINDOW [resourceID] ;

CLOSE WINDOW[resourceID] ;

FRONT WINDOW [resourceID] ;

BACK WINDOW [resourceID] ;

This results in calls to low level WINDOW handling routines in the System Calls shell file.

RE: FRONT and BACK, although code is generated for these, it is my understanding that their use in OSU is on hold at the present time.

EG:

OPEN WINDOW [328] ;

CLOSE WINDOW[328] ;

FRONT WINDOW [328] ;

BACK WINDOW [328] ;

iv) SAMPLE SEQUENCE COMMAND FILE

This is a sample Sequence Command file. Comment lines have been added to identify the various sections of the file. Also a comment has been added to identify each verb - object pairing the first time it appears. The comment lines are those which start with an asterisk. The actual sequence statements have been printed in bold face type. This is not an exhaustive sample of all verb - object pairings but represents a representational file that was used extensively as a test file during development.

```
* the initiaization block.
ITEMHIT = INIT ;
  * a call to a procedure name procall1.
   DO  [procall1] ;
   * the cursor's appearance is to be changed to an I-Beam.
   CHANGE CURSOR [Ibeam] ;
  * a call to an alert ID number 303, of type Stop alert.
   OPEN ALERT [303=a1] ;
  * a call to a dialog. ID number 204.
   OPEN DIALOG [204]  ;
  * string ID 101 is to be inserted in paramtext position 4, Dialog 204
  * will need to be made a procedure.
   INSERTSTR 4 [101] ;
  * an itemhit block for the first control in alert 303's ditl list. It is a
  * check box and all sequence statements in this block will be included
  * in  the case for this control. Also this will cause alert 303 to be
  * made a procedure.
   ITEMHIT = DITLITEM [1=a303=c] ;
```

```
        CHANGE CURSOR [arrow] ;
    * end of the ditl itemhit block.
    . ;
    * menu item extra2, item #3 in menu list #3, will be checked.
    CHECK extra2 [3=3] ;
    CHECK extra three [3=4] ;
* end of the initialization block.
. ;

* start of the menubar block.
ITEMHIT = MENUBAR ;
  * an itemhit block for menu list demo, list #2.
   ITEMHIT =  Demo [2=0] ;
   * an itemhit block for menu item aDialog, item #1 in menu list #2.
   * all code for sequence statements in this block will be included in
   * the procedure generated for menu item aDialog.
      ITEMHIT = aDialog [2=1] ;
          OPEN DIALOG [201] ;
          INSERTSTR 3 [] ;
          ITEMHIT = DITLITEM [1=D201=p] ;
          * enable menu item this, item #1 in list #24.
              ENABLE this [24=1] ;
              OPEN DIALOG [202]  ;
          . ;
           ITEMHIT = DITLITEM [3=D202=p] ;
              ENABLE this [24=1] ;
              OPEN DIALOG [201]  ;
```

```
        OPEN ALERT [302=a3]  ;
      * close dialog #202. Since this is in a control block that will
      * be inserted in the procedure for dialog #202, this will
      * result in an exit from that procedure.
        CLOSE DIALOG [202]  ;
      . ;
      ITEMHIT = DITLITEM [3=D204=p] ;
        CHECK this [24=1] ;
        OPEN ALERT [303=a2]  ;
      . ;
      ITEMHIT = DITLITEM [2=D201=b] ;
        CLOSE DIALOG [201] ;
      * disable menu item this, item #1 in menu list #24.
        DISABLE this [24=1] ;
        OPEN ALERT [302=a3]  ;
      . ;
* end of the itemhit block for menu item aDialog, item #1
* in menu  list #2.
  . ;
  ITEMHIT = Quit [2=3] ;
   * this will cause the entire application to be terminated.
    QUIT ;
  . ;
* end of the itemhit block for menu list demo, list #2.
 . ;
ITEMHIT =  What next [24=0] ;
  ITEMHIT = that [24=2] ;
```

```
        OPEN ALERT [302=a3] ;
        ITEMHIT = DITLITEM [1=a302=r] ;
            CHANGE CURSOR [Cross] ;
        . ;
        CHANGE CURSOR [Watch] ;
        DISABLE this [24=1] ;
        CHECK the [24=3] ;
        DISABLE other [24=4] ;
      * uncheck menu item extra2, item #3 in menu list #3.
        UNCHECK extra2 [3=3] ;
        UNCHECK extra three [3=4] ;
      . ;
    . ;
  * end of the menubar block.
. ;
```

v) SAMPLE SEQUENCE CONTROL FILE

The first number on each line is the type of the object, 4 is a dialog and 5 is an alert. A window would be a 3. The second number is the ID number of the object. The last number is the count for the ditl list for that object.

```
4 200 2
4 201 3
4 202 5
4 203 2
5 303 3
5 302 2
5 301 2
5 300 2
```

vi) USES RELATIONSHIP CHART FOR OSU PROJECT [Handloser 88]

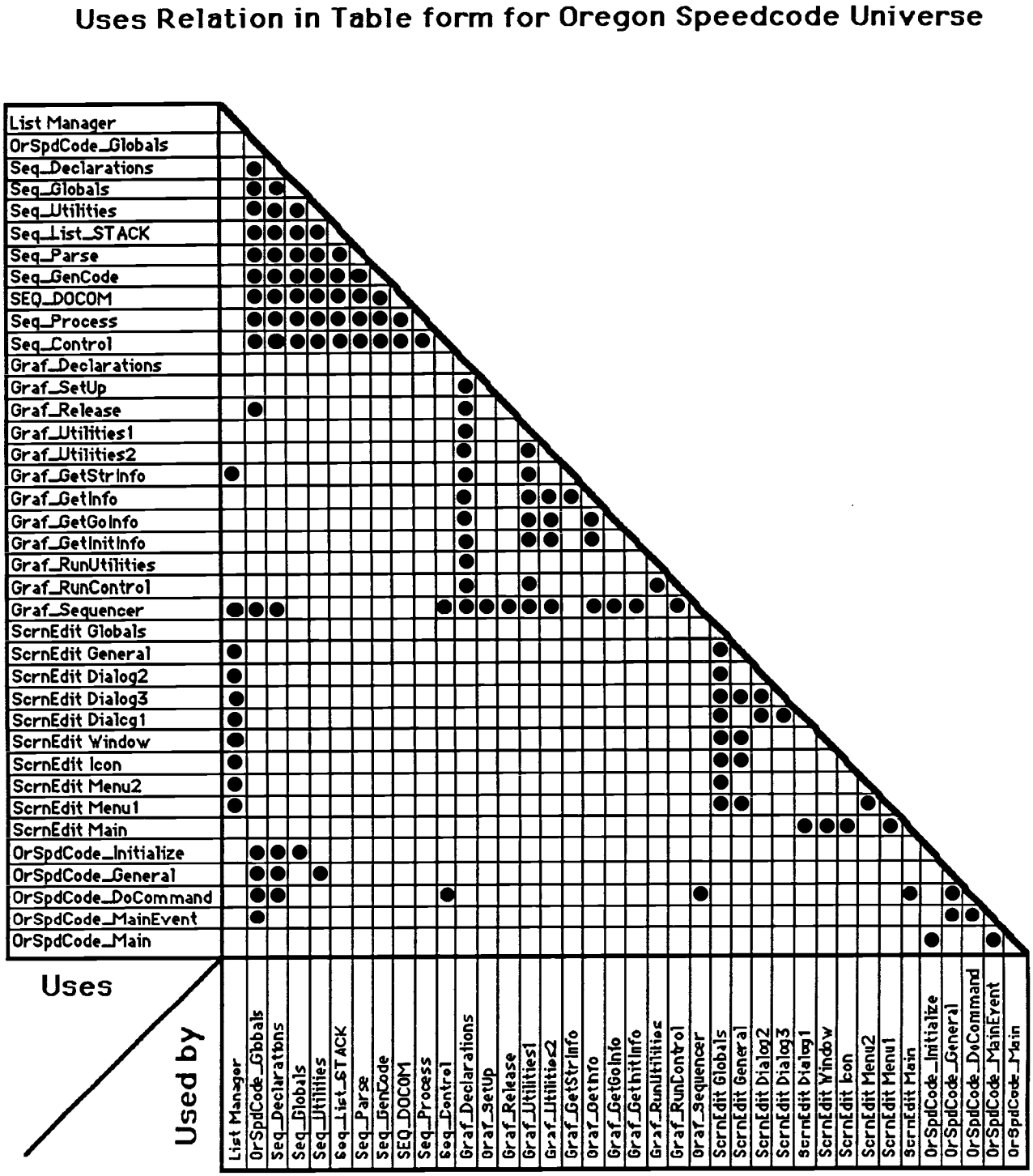

Figure 23. Uses Hierarchy for OSU