FORMAL GRAMMAR SPECIFICATIONS OF USER INTERFACE PROCESSES

by

MICHAEL WAYNE BATES

Bachelor of Science in Arts and Sciences

Oklahoma State University

Stillwater, Oklahoma

1982

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 1984

FORMAL GRAMMAR SPECIFICATIONS OF

USER INTERFACE PROCESSES

Thesis Approved:

M. J. Folk

Thesis Adviser

W. E. Hedrick

J P Chandler

Norman N. Durham

Dean of the Graduate College

PREFACE

The benefits and drawbacks of using a formal grammar model to specify a user interface has been the primary focus of this study. In particular, the regular grammar and context-free grammar models have been examined for their relative strengths and weaknesses.

The earliest motivation for this study was provided by Dr. James R. VanDoren at TMS Inc. This thesis grew out of a discussion about the difficulties of designing an interface that TMS was working on.

I would like to express my gratitude to my major advisor, Dr. Mike Folk for his guidance and invaluable help during this study. I would also like to thank Dr. G. E. Hedrick and Dr. J. P. Chandler for serving on my graduate committee.

A special thanks goes to my wife, Susan, for her patience and understanding throughout my graduate studies.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

For some time now, there has been a general trend toward interactive computing. This is due in part to the decreased costs of powerful terminals and computing hardware. As the interactive systems become more and more complex, there are increased demands of the user interface for these systems.

Interactive systems have typically suffered from the inability of designers to clearly visualize and easily express their designs as concrete, comprehensible models [3]. A common method for constructing the human-computer interface is by ad hoc techniques. After an ad hoc system has been implemented, attempting to correct deficiencies or make minor design changes are often difficult because the original design was unclear or incomplete. A tool for writing a clear and complete specification of large or complex user interfaces would be a useful item.

If one looks from the right angle, a great deal of similarity can be seen between the function of a compiler and the function of a user interface. The compiler recognizes a legal sequence of symbols and performs actions, such as generating code, when certain substrings of legal symbols

are recognized. The user interface for a software system also accepts a legal string of input symbols and performs actions based on the recognition of legal substrings. The only difference is that the user enters the symbols interactively from a terminal one at a time to the user interface. The input symbols to an interface may be typed commands, function key presses, joystick motions or other user input that can be broken down into discrete entries.

Compiler specifications have been based on formal grammars for some time. The grammar describes the rules for valid program constructions in a rigorous and nonambiguous manner. It also provides an accurate means of communication of a specification between two or more people.

It seems that many of the benefits of using a formal grammar to specify a compiler would also apply to the specification of an interactive user interface. The interface should be complete, consistent and unambiguous. It should also be constructed in such a way that it could be understood and modified by persons other than the original designer(s).

If formal grammars were used to specify a user interface and a recognizer were built (either by hand or mechanically) to accept the language specified by the grammar, certain practical limitations to the recognizer should be understood. For instance, input string of tokens and the associated actions are generated in an interactive manner. This implies that "backing up" in the input stream to try to

reprocess a token may be difficult or even impossible. It also implies that no lookahead may be performed in the input string. These problems may be overcome by choosing a suitable type of recognizer and building it with these limitations in mind.

There are four classes of formal grammars in the Chomsky hierarchy of grammars; regular grammars, context-free grammars, context-sensitive grammars, and unrestricted grammars. In the hierarchy, each class of grammars contains all of the previous class of grammars. For example, all regular grammars are context-free but not all context-free grammars are regular, all context-free grammars are context-sensitive but not all context-sensitive grammars are context-free, and so on. A regular grammar can be recognized by a deterministic finite state machine and a context-free grammar can be recognized by a pushdown automaton. These two classes of grammars are well understood and should be powerful enough to specify a very complex user interface.

Most of the previous work done in the area of using grammars to specify user interfaces [2, 3, 4, 6, 8, 10, 11, 12, 13] has approached the problem by finding an application and then applying a grammar to it. There has been little discussion (with the exception of Jacob [6]) of why a particular type of grammar was chosen; only that a grammar was chosen. The orientation of this study is not toward a particular application, so the grammar models themselves are

examined for the types of interfaces they would be most suited to describe.

This study is a two-part investigation of the use of formal grammars for the specification of interactive user interfaces. The first part is a study of the relative benefits and drawbacks to using a formal grammar to specify an interface. Some of the benefits examined include the use of a formal grammar as an analytical tool, the capability to automatically manipulate the grammar, and the capability to develop fast prototypes of the interface. Another benefit studied is the ability of the grammar to specify the user interface at more than one level of detail with the same language.

The drawbacks of using a formal grammar for the specification of a user interface are also studied. The grammar takes time to construct and for large or complex interfaces may be quite large. Other potential difficulties, such as backing up in the parser, are also discussed.

The second part of this study is a discussion of the relative strengths and weaknesses of two particular formal grammar models: the regular grammar model and the context-free grammar model. The regular grammar, being the simpler of the two models, is easier to implement. However, the context-free model is the more powerful model and is capable of describing a more complex interface. This study draws these and other distinctions between the two models and

examines some types of user interfaces where each model might be appropriate.

Finally, some conclusions are drawn about the use of formal grammars to specify user interfaces. The benefits are weighed against the drawbacks of using a grammar based specification and the relative strengths and weaknesses of the regular and context-free grammar models are assesed.

CHAPTER II

AN OVERVIEW OF FORMAL LANGUAGE THEORY

Introduction

It seems appropriate at this point to examine some basic concepts of formal language theory, particularly those concepts that may apply to the specification of a user interface. The reader that is familiar with formal language theory is invited to skim this chapter observing only the notation used in the description and the examples. This discussion will be limited to the Chomsky hierarchy of grammars and their associated recognizers.

A language L will be defined to be a set of finite length strings over some finite alphabet Σ. If there were only five strings in a particular language, the strings could be listed and the language would be completely specified. However, useful languages such as those that specify a programming language or a user interface are rarely describable by a small or even finite number of strings. Normally, we will want the specification of a language to be of finite size, even if the number of strings in the language is infinite. This chapter presents two methods for specifying a language. One method generates a language and the other recognizes a language.

Grammars

One method for specifying a language is to use a fixed starting point and a set of rules for forming the strings in the language. This generative system is called a grammar. Aho and Ullman provide a common definition for a grammar:

A grammar is a 4-tuple $G = (N, \Sigma, P, S)$ where

1. N is a finite set of nonterminal symbols (sometimes called variables or syntactic categories).

2. Σ is a finite set of terminal symbols, disjoint from N.

3. P is a finite subset of

$$(N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$$

An element (a,b) in P will be written a -> b and called a production.

4. S is a distinguished symbol in N called the sentence or start symbol [1, p. 232].

An example grammar is shown in Figure 1. The nonterminal symbols are S and A and the terminal symbols are 0 and 1. S is the start symbol and e represents the empty string.

G = ({S,A}, {0,1}, P, S) where P consists of

S -> 0A1
0A -> 00A1
0A1 -> 01

Figure 1. An Unrestricted Grammar

This grammar generates strings of the form 01, 0011, 000111, and so on indefinitely. The grammar defines this language of an infinite number of strings in a finite manner by using recursion within the productions. The second production for example, contains the nonterminal A on both sides of the arrow.

The Chomsky hierarchy classifies grammars by placing restrictions on the format of their productions. A grammar with no restrictions is called an unrestricted grammar. Productions of the form A -> B, where A and B are contained in $(N \cup \Sigma)^*$ are allowed. The grammar in Figure 1 is an example of an unrestricted grammar.

Context-sensitive grammars form a proper subset of unrestricted grammars. A context-sensitive grammar may only have productions of the form

1. a -> b, where a and b are contained in $(N \cup \Sigma)^*$ and the length of a is less than or equal to the length of b ($|a| <= |b|$).

2. S -> e, where S is the start symbol and e is the empty string.

An example of a context-sensitive grammar is shown in Figure 2.

G = ({S,A}, {0,1}, P, S) where P consists of

S -> A
S -> 0A1
0A1 -> 00A11

Figure 2. A Context-Sensitive Grammar

The grammar of Figure 2 generates strings of the form 01, 0011, 000111 and so on exactly as the unrestricted grammar of Figure 1 does. The term context-sensitive comes from rules like aBc -> aZc. The B may only be rewritten when in the context of a and c.

The next most restrictive type of grammar is the context-free grammar. This grammar contains the added restriction that productions are limited to the form:

1. a -> b, where a is a single nonterminal symbol and b may be any combination of terminal or nonterminal symbols.

Figure 3 shows a context-free grammar that generates the same strings as Figures 1 and 2 (01, 0011, 000111, ...).

G = ({S,A}, {0,1}, P, S) where P consists of

S -> A
A -> 0A1
A -> 01

Figure 3. A Context-Free Grammar

The right-linear or regular grammar is the most restrictive type of grammar in the Chomsky hierarchy. Productions of this grammar are limited to the form:

1. A -> xB
2. A -> x

where A and B are nonterminal symbols and x is contained in Σ^*.

The language example defined by the three previous grammars (01, 0011, 000111, ...) cannot be defined by a regular grammar. Because of restrictions applied to the productions, the regular grammar is not powerful enough to describe it. The proof of this is simple but not within the scope of this study. An example of a regular grammar that generates the language $\{0^i1^j \mid i,j > 0\}$ is shown in Figure 4.

G = ({S,A}, {0,1}, P, S) where P consists of

S -> 0S
S -> 0A
A -> 1A
A -> 1

Figure 4. A Regular Grammar

An example of a metalanguage for a grammatical description is the Backus Naur Form, particularly for context-free grammars. In Backus Naur Form, or BNF, the production symbol "->" is represented by "::=". Nonterminal symbols are represented by strings enclosed in angled brackets, "<"

and ">". Terminal symbols are directly represented as themselves. The vertical bar, "|", may be used to separate alternative right hand sides for a single left hand side. Figure 5 shows a sample grammar written in Backus Naur Form.

```
<S> ::= <A>
<A> ::= 0<A> | 01
```

Figure 5. A BNF Grammar

The BNF grammar description is well known and easily readable by anybody with a background in compiler design or language theory. It is also easy to learn by those who have never seen it before but want to use it.

Recognizers

Another way to specify a language is to define a tool to recognize it. The language is then defined to be the set of all strings recognized (accepted) by the tool. Just as there are several classes of grammars with varying power for specifying the different types of languages, there are several classes of recognizing tools for recognizing the same languages.

A recognizer can be viewed as being made up of three parts: an input/output tape, a finite state control, and

some form of auxiliary memory. Figure 6 shows a logical description of a recognizer.

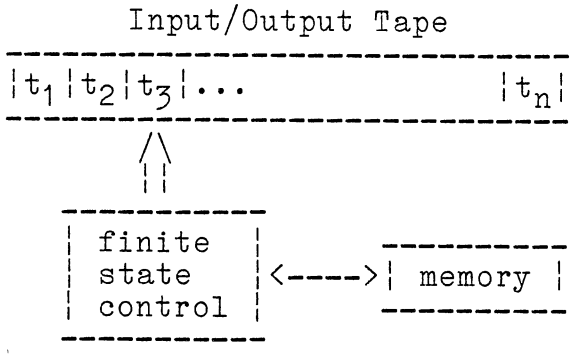

Figure 6. A Recognizer

The input/output tape is a sequence of input symbols (tokens) from a finite alphabet. The current symbol is used by the finite state control to determine what to do next. The tape can be repositioned to the left or right, one symbol at a time by the recognizer. The finite state control controls the positioning of the tape and the contents of the auxiliary memory. It can be thought of as a program that performs the recognition by controlling the sequence of moves it makes.

A configuration of a recognizer includes the state of the finite state control, the input/output tape with a marker at the current symbol, and the contents of the auxiliary memory. A move is a transition from one configuration to the next. If there is only one possible move from

each configuration for any given input symbol, the recognizer is said to be deterministic.

The most general class of recognizers is the Turing machine. It recognizes the class of language definable by an unrestricted grammar. The Turing machine may be deterministic or nondeterministic. The input/output tape may be of infinite length, but the number of nonblank symbols on the tape must be finite. The tape may be read from or written to and may be repositioned one symbol to the left or right as dictated by the finite state control. The Turing machine halts when it reaches a configuration for which no move is defined for the current input symbol. The nonblank portion of the input/output tape is the output of the machine.

If the potentially infinite input/output tape of a Turing machine is limited to a finite length, a linear bounded automaton is the result. A nondeterministic linear bounded automaton is capable of recognizing the class of languages definable by a context-sensitive grammar.

This study is most interested in two other types of recognizers: the pushdown automaton and the finite state machine. The pushdown automaton is a recognizer with a read-only input tape, a finite state control, and a pushdown stack for auxiliary memory. A nondeterministic pushdown automaton recognizes the class of context-free languages. However, in order to recognize or reject a string in linear time, a subset of context-free languages is usually used.

This subset is composed of deterministic context-free languages. These are recognized by a deterministic pushdown automata.

The pushdown automaton uses the current input symbol, the contents of the top element of the stack memory, and the current state of the finite state control to determine an appropriate move. Certain types of pushdown automata are allowed to look ahead a fixed number of symbols in the input tape to assist in making the correct move. Note that for user interfaces this may be impossible because of tokens the user has not yet entered.

A finite state machine is equivalent to a pushdown automaton without the pushdown stack. It uses only the current input symbol and the current state of the finite state control to make a move to the next state. Finite state machines recognize regular sets, or languages defined by regular grammars. A graphical representation of a finite state machine is shown in Figure 7. In the graphical representation, the /\ defines the start state, the ◎ defines the final states and each labeled arc defines a transition for the given symbol to the given state. The machine depicted in Figure 7 recognizes the language generated by the grammar in Figure 4.

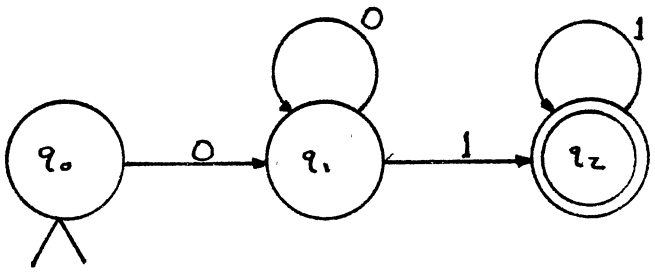

Figure 7. A Finite State Machine

Normally, there is more work to be done by a recognizer, or parser, than simply to accept a string as part of a language. Certain "actions" are usually performed along the way as the input string is being parsed. The actions might be associated with code generation if the parser were part of a compiler for a programming language. If the parser were part of a user interface, the actions might be screen formatting and menu selection.

A pushdown automaton gased on a grammar that generates a context-free language can be used to parse a string of input tokens. A "bottom-up" parse is one in which the input tokens are shifted onto the stack until the right hand side of a production rule is recognized. The details of how this is done are beyond the scope of this paper. The symbols from the right hand side of the recognized rule are then popped from the top of the stack and the nonterminal symbol

from the left hand side is placed on top of the stack in their place. This is called a rule reduction and action associated with that rule is performed at this time. The process continues until the end of the input string is reached. At that point if the finite state control is in a final state and the stack is empty, the string is accepted as part of the language. If the parser reaches a point where there is no move from a particular configuration for an input token, the input string is rejected as not in the language and some type of error recovery may be performed.

Parsing a string in a regular language is a simpler process. The ony two pieces of information the finite state control has to act on is the current input symbol and the current state of the machine. From these the parser can deduce the next state of the finite state control. Any action to be performed by the finite state machine is associated with each input symbol.

Summary

This chapter presented a simplified look at the Chomsky hierarchy of grammars and their associated recognizers. A more complete treatment of grammars and recognizers can be found in [1, 5]. This study is most interested in context-free and regular grammars with their respective recognizers. The primary difference between the two is the primitive form of memory the context-free languages allow that the regular languages do not. The finite state machine

can only determine where it is while the pushdown automaton can determine something about how it got there.

CHAPTER III

USING FORMAL GRAMMARS TO SPECIFY USER INTERFACES

Introduction

This chapter describes the use of formal grammar models to specify a user interface. The first objective is to describe what a user interface is. The description will be more by example than by rule, since there are no hard rules to define a user interface.

Next is a discussion of the benefits of using a formal grammar model to specify a user interface. The utility of a grammar to specify an interface at several levels of detail with the same language and the prospect for fast prototyping are among the benefits. Several other benefits are also discussed in this chapter.

Finally, an examination of some reasons why a formal grammar might not be the ultimate choice for a specification tool. The potential large size of the grammar and the time it takes to construct it are only two of the drawbacks studied.

Definition of a User Interface

Separating the user interface from the underlying software system is usually a difficult task, particularly when it is implemented in an ad hoc manner. The user interface is so closely intermixed with the software system that it is sometimes difficult to tell whether a piece of code belongs to the interface or the software system itself. For this reason, the user interface is difficult to define by any other means than the functions it performs.

A compiler for a high level language is certainly a type of user interface. A user would like to instruct the computer to perform a task, but the thought of creating a sequence of machine readable zeros and ones is not very appealing. The user would prefer to write instructions in a high level language and allow the compiler to generate machine readable code, or at least code that could be automatically converted to machine readable form. The compiler is the interface between the user writing in a high level language and the computer that can only use zeros and ones. Compilers have been specified by formal grammars for a long time.

Reisner [12] uses the term "action language" to differentiate between a programming language and a language describing a user interface for her graphics user interface application. The action language may be defined as the sequence of button pushes, joystick motions, typing actions, etc., that are performed by a user interacting with a

terminal. A close parallel exists between the high level language described by a formal grammar in a compiler and the action language that can also be described by a grammar in a user interface.

Lawson, Bertran and Sanagustin suggest two functions the user interface for an interactive system should perform.

> Firstly it has to produce a sequence of messages which are understandable to the computer system and which correspond to the function specified by the sequence of buttons depressed by the user. This sequence of keys will in general define an 'operation' and various 'operands', i.e. a 'function' to be performed on or with some 'variables'.
>
> A second function of [a user] interface is to detect erroneous sequences of buttons thus preventing them from reaching the [software] system. The principal types of errors to be detected are the syntactic errors i.e. incorrect ordering, invalid button combination, etc. Of course, there are other types of errors which the [software] system itself has to detect [8, p. 52].

A semantic error, such as requesting a record that is not in the database, is an example of a type of error the software system itself would have to detect.

A third function of a user interface should be added. That is the management of the display screen and the input devices. The interface formats the display with menus or prompts and accepts all input. Any prompting for additional information is also handled by the user interface.

For the purposes of this study, a user interface will be defined as a piece of software that recognizes and accepts an action language and performs the kind of functions

described above. The emphasis of this study is on interactive user interfaces which grammars have not been commonly used to describe. These are the interfaces between a user sitting at an interactive terminal and a software system performing the tasks the user requests. The software system may be a database retrieval system, a graphics system, or some other interactive system.

Benefits of a Formal Model

As software systems grow larger and more complex the ability of one person to fully comprehend the entire system becomes limited. The same holds true for user interfaces. When more than one person is involved in the design of a user interface the problems of communicating ideas and maintaining consistency arise. Jacob [6] makes the argument that one is handicapped in trying to design a good user interface without a clear and precise technique for specifying such an interface. A formal grammar provides a concrete model for specifying the user interface.

One very useful property of a formal grammar is its ability to specify an interface at more than one level of detail with the same language. For a high level specification, some nonterminal symbols may not be defined in terms of the terminal and nonterminal symbols they represent. These undefined nonterminal symbols might represent a class of terminal symbols whose actual makeup may as yet be undetermined.

Figures 8 and 9 are two different descriptions for a hypothetical user interface for a student grade keeping system. Figure 8 is a state transition diagram representation and Figure 9 is a BNF description. The example depicted by Figures 8 and 9 contains several undefined nonterminal symbols. Among them are the symbols "<id>", "<cid>", and "<sid>". Each undefined symbol, such as "<id>", represents one of a set of identifier symbols. The specification of the contents of the set of undefined symbols may be deferred to a later time.

The undefined symbols "<browse_class>", "<browse_student>", <edit_class>", and "<edit_student>" are not defined for a different reason than the symbols above. These symbols represent complete subsystems within the user interface. That is, the symbol "<browse_class>" may expand into a large piece of the user interface at the lowest level. However, the details of the commands that browse a class are not necessary for a specification at this level. The lower level specification that might include an expanded version of "<browse_class>" would use the same metalanguage for description as that used by the higher level specification.

It is conceivable that a low level specification might not ever define a particular nonterminal symbol. There are cases where it might be better not to fully define a symbol. The symbols "<good_pw>" and "<bad_pw>" in Figures 8 and 9 are examples of such a case. There is a semantic relationship between an id and a correct password that

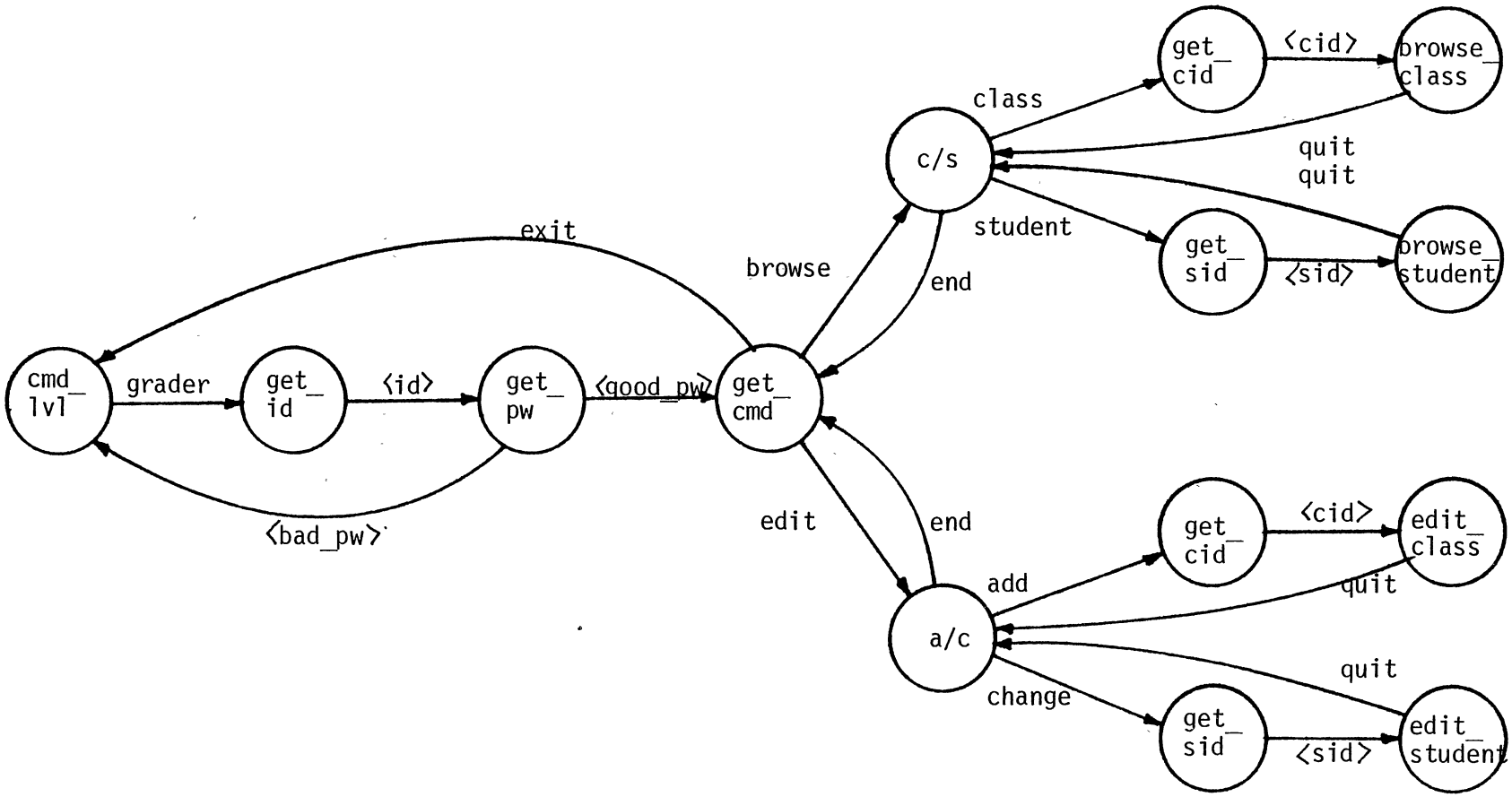

Figure 8. State Transition Diagram for a Hypothetical User Interface.

```
<grade_prog> ::= grader <id> <bad_pw>                          (1)
               |  grader <id> <good_pw> <get_cmd>               (2)

<get_cmd>    ::= browse <c/s> exit                             (3)
               |  browse <c/s> <get_cmd>                       (4)
               |  edit <a/c> exit                              (5)
               |  edit <a/c> <get_cmd>                         (6)

<c/s>        ::= class <cid> <browse_class> quit end           (7)
               |  class <cid> <browse_class> quit <c/s>        (8)
               |  student <sid> <browse_student>
                  quit end                                     (9)
               |  student <sid> <browse_student>
                  quit <c/s>                                   (10)

<a/c>        ::= add <cid> <edit_class> quit end               (11)
               |  add <cid> <edit_class> quit <a/c>            (12)
               |  change <sid> <edit_student> quit end         (13)
               |  change <sid> <edit_student> quit <a/c>       (14)
```

Figure 9. A BNF Description of a User Interface for a Grade Keeping System.

cannot be specified syntactically unless the id and the password are fixed and cannot be changed. Pragmatically, fixing the ids and passwords is probably not a good idea.

If an interface needed to be modified a year or more after it was designed, the design documentation might be some of the most important information the person modifying it needs. This is especially true if the person modifying the interface was not involved in the original design. The potential to overlook documentation during the design stage of a user interface is always present. A formal grammar description of the design in a common notation, such as BNF, could prove valuable. When a modification is made to the interface and the grammar, the same benefits that applied to the initial design are still valid, such as early checking for consistency and precisness. In addition, the person modifying the system is less likely to upset the entire system because of an unknown side-effect with a clear and concise design to look at.

Reisner [12] cites several other useful properties of a formal grammar description for a user interface including the following four:

1. A major value of a formal description of a user interface is its use as an analytical tool. A paper and pencil representation of an interface can be analyzed much earlier than a working model. An early analysis may turn up design flaws before they are implemented, when they are easier to correct.

2. The fact that a formal model forces the interface designer to be precise about the specified system can be of value. Again, the early detection of an ambiguity or an unexpected case is easier to correct at design time.

3. A uniform method for detecting syntactic errors is accomplished by a formal description. When errors occur they can be detected and dealt with consistently.

The interactive nature of the input allows the parser the luxury of stopping at each syntactic error to allow the user to fix the problem before going on. The parser detects an error when there is no valid move defined from the current configuration of the recognizer for the next input token. If an error of this type occurs with a compiler parsing a program, the compiler must flag the error and fix itself in such a way that it can continue to parse the program. For some simple errors the compiler may be able to determine a correction that will allow it to continue generating output code. Most errors, though, are significant enough to force the parser to repair itself only well enough for it to continue syntax checking. The compiler can hardly stop to ask the user to fix an error so it can go on generating code, especially if it is running detached from a terminal. However, a parser for an interactive user interface can stop any time it finds a syntactic error and allow the user to fix it so it can go on doing useful work.

4. A formal grammar may be automatically manipulated. For example, it should be possible to construct and examine

any desired combination of strings accepted in the language for testing the relative difficulty or awkwardness of an interface.

Using tools such as LEX [9] and YACC [7] in the UNIX [13] operating system, a parser may be constructed almost directly from a BNF grammar description. LEX automatically builds a recognizer for regular grammars that can be used by itself or to recognize tokens in combination with a parser generated by YACC. YACC builds a recognizer for a large class of context-free languages. A substantial portion of the input to YACC is a BNF grammar. The parser generated by YACC could be the basis for a prototype of the user interface.

Prototyping allows one to put together quickly an interactive "slide show" where the user can see the various static responses to typed input commands. For example, a prototype of a menu-driven student grade keeping system could have a set of screens that contain all of the menus and some screens that contain browse or edit request. A user would be allowed to select menu options, browse, or edit while the parser was recognizing the input and causing the appropriate static screen to be displayed. The parser can be viewed as the control for a random access slide projector. A prototype of this type could be valuable for discovering weaknesses or inherently awkward portions of the interface. This type of slide show might also be useful in

demonstrating ideas about the interface in a way that could not be demonstrated as effectively on paper.

The issue of prototyping alone may be enough to justify the time and effort to construct a grammar for the user interface. A developing system is subject to numerous design changes and the ability to modify a prototype quickly and accurately could have a very positive effect on the value of the prototype itself.

Drawbacks of a Formal Model

There are tradeoffs when using a formal grammar to describe a user interface. It takes time and some skill to construct a grammar. For a complex user interface the grammar may be quite large if all legal strings are described.

Reisner [12] points out that a straightforward BNF-like notation might not be the ultimate choice. A notation is desired that can 1) describe all and only all of the legal strings for the user and 2) show the structure of the language with as few nearly redundant rules as possible. These turn out to be somewhat contradictory requirements unless the size of the grammar is reduced by introducing some semantic restrictions to prevent illegal strings.

A grammar that describes all and only the legal strings for an application may contain a large number of nearly redundant rules. For example, the student grade keeping interface descriptions in Figures 8 and 9 could become cluttered if all correct combinations of an id and a password

were included in the description. Rule (2) in Figure 9 might expand to something like the rules shown in Figure 10. Adding a semantic restriction that the id and password must correspond to each other would not reduce the number of grammar rules. However, as shown in Figure 11, it would generally clean up the structure of the grammar to more clearly show the language accepted.

Another potential difficulty is observed by Anson [2]. In certain cases it could be difficult for grammar to adequately represent the features of a real interactive device. A device such as a knob that must retain its value between uses and which can be changed by the user at any time is difficult to be simulated by a grammar. If the value is to be retained it must be stored outside the grammar and it can only be changed when the interface requests a new value.

For example, the grade keeping user interface might allow input from a switch on the front of the terminal to adjust the brevity of error messages. The problem of storing the current value of the switch (up or down) can be solved by storing the value in an external variable. The diagram in Figure 8 would change to look like Figure 12. The rules of Figure 9 would also expand. The first few are shown in Figure 13.

The point to be made here is that there are some types of input to a user interface that are very difficult to describe with a grammar. When there is a global value that

```
<grade_prog> ::= grader mike hiccup <get_cmd>
               | grader john dog    <get_cmd>
               | grader mary lamb   <get_cmd>
               | grader peter piper <get_cmd>
               | grader joe ball    <get_cmd>
               | ...
```

Figure 10. Expanded Rule (2) From Figure 9.

```
<grade_prog> ::= grader <id> <good_pw> <get_cmd>

<id>         ::= mike
               | john
               | mary
               | peter
               | joe
               | ...

<good_pw>    ::= dog
               | cat
               | lamb
               | hiccup
               | ball
               | ...
```

Figure 11. Figure 10 Restructured With Semantic Restrictions

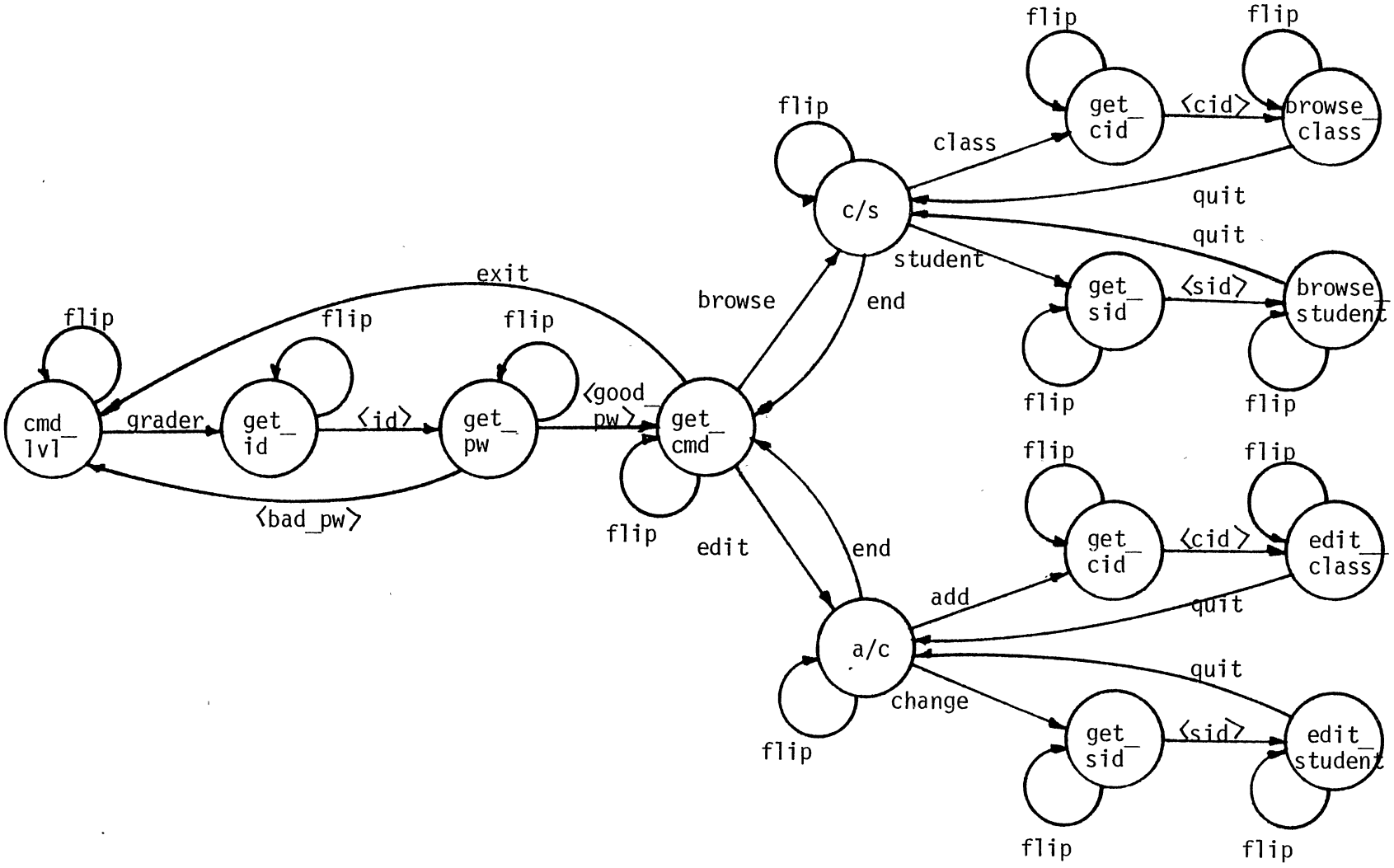

Figure 12. State Transition Diagram Modified for a Switch.

```
<grade_prog> ::=  <switch> grader <switch> <1d>
                  <switch> <bad_pw>
                |
                  <switch> grader <switch> <1d>
                  <switch> [good_pw> <switch> <get_cmd>

<get_cmd> ::=     <switch> browse <switch> <c/s>
                  <switch> exit
                |
                  <switch> browse <switch> <c/s>
                  <switch> <get_cmd>
              ...

<switch>      ::= flip
                | flip <switch>
                | e (empty string)
```

Figure 13. BNF Description Expanded for a Switch

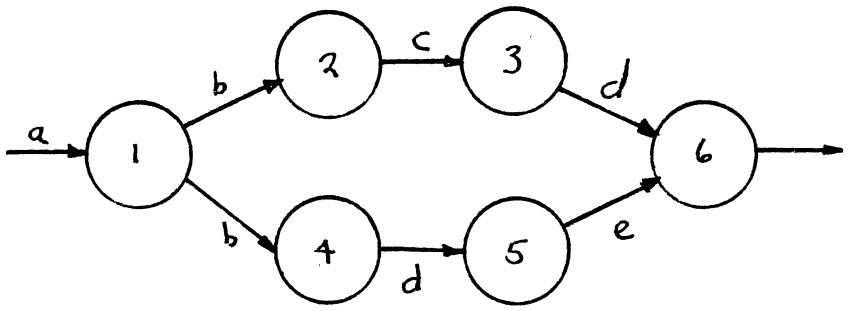

Figure 14. A Nondeterministic Grammar Description

can be changed at any time by the user, a grammatical description may not be sufficient.

Another potential difficulty with a grammatical description of an interactive interface is the introduction of nondeterminism into the grammar. There are deterministic parsers that simulate nondeterministic parsers to parse a nondeterministic grammar. However, the parser backtracks to try another path if a "dead end" is reached. Figure 14 shows a fragment of an example grammar that recognizes the strings "...abcd..." and "...abde...". The grammar is nondeterministic because there is more than one path from state 1 that is labeled with a "b". If a backtracking parser was trying to recognize the string "...abde...", the parser might arbitrarily choose the path to state 2 when the "b" is encountered and perform the action associated with state 2. When the "d" is encountered, the parser cannot make a move from state 2 for a "d" so it must backup to state 1 and start over on the path to state 4 with the "b". One of the two potential problems with this backtrack in an interactive system are that some of the input must be saved so that it can be rescanned for a different path. This is not as difficult to overcome as the second problem which is that the action associated with state 2 (the wrong path) may have generated output to the user. Hanau points out this problem and states:

> While a compiler may reasonably 'pop' symbols added to a symbol table on such a false path, an

> interactive system can hardly advise the user to 'ignore all messages since...' [4, p. 276].

This does not imply that backing up in the parser is impossible, but it may be difficult because of the actions performed.

CHAPTER IV

THE REGULAR AND CONTEXT-FREE GRAMMAR MODELS

Introduction

Two grammar models seem to be particulary suited for the specification of a user interface. The two grammar models are the regular and context-free grammars.

Previous work in the area of using formal grammars to specify user interfaces has typically been concerned only with the benefits and drawbacks of using grammars for the specification. There has been little discussion of why a particular grammar was chosen for an application, only that it was chosen. A notable exception to this is a paper by Jacob [6].

The focus of this study is not on a specific application, but on the grammars themselves. The set of possible choices of grammars to use for a specification that range from the simplest regular grammar to the most powerful unrestricted grammar. However, because of practical space and time restrictions, the regular and the context-free grammars are usually the two models to choose from.

The Regular Grammar Model

The simplest grammar model for specifying a user interface discussed in this study is the regular grammar. The language generated by a regular grammar is recognizable by a finite state machine. The finite state machine itself is a compact recognizer which requires no extra space for memory and runs in linear time.

Typically, the regular grammar is depicted as a state transition diagram. Each transition is associated with the input of a token from the user. An action to be performed is also associated with each transition. Whenever the transition occurs, the system performs the action. The state diagram contains an explicit sequence that is implicit in a BNF description. The explicit sequence description makes it easy for the designer to specify when actions are to occur without any modification to the grammar to provide this control. The diagram in Figure 8 of the previous chapter demonstrates a regular grammar for a hypothetical user interface.

The state transition diagram can also aid in top down design by adding the concept of a "complex state" [3]. A complex state is formed by substituting a nonterminal symbol for a section of the diagram that contains one entry and one exit. The nonterminal symbol is then described by its own state transition diagram. Figure 15 shows the grammar of Figure 8 which has been modified to use complex states.

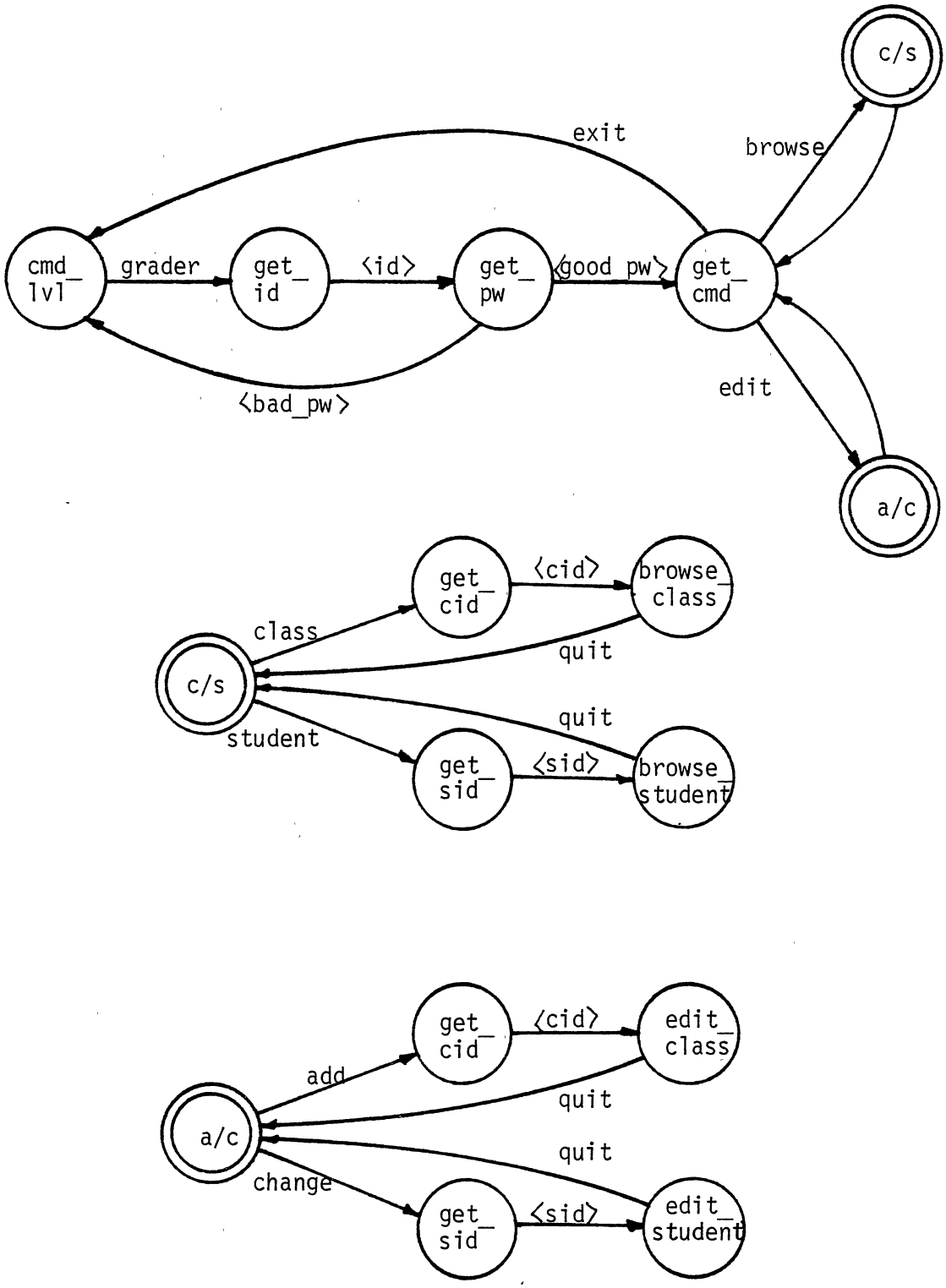

Figure 15. Figure 8 Modified for Complex States.

The primary drawback to the regular grammar model is that it is not powerful enough to specify certain constructions in a user interface. For example, the grammar represented in Figure 16 shows the grammar of Figure 8, which has been modified to allow the user to suspend the browsing of the class to browse another class an arbitrary number of times. Each time the user types "resume", the next previous suspended browsing operation is resumed.

There are at least two methods for making the regular grammar model work for an application that requires nested recursive calls and the ability to back out to the top level again.

1. If it is reasonable to limit the number of recursive invocations to a finite and reasonably small number, the grammar may be "linearized" to form a possibly large but regular grammar. In a linearized grammar, each recursive call is represented by a new state with the transition to that state labeled with the calling token. There is also a transition to the previous state to allow the interface to back out. Figure 17 shows an example of part of a state transition diagram that allows recursion to nest at most two levels deep. The linearization of the diagram acts as a counting mechanism for the grammar.

2. It may not be reasonable to limit the recursion to a small, finite depth. If a regular grammar is still desired, it can be used if the rules are bent slightly. A simple static variable to act as a counter can be added to

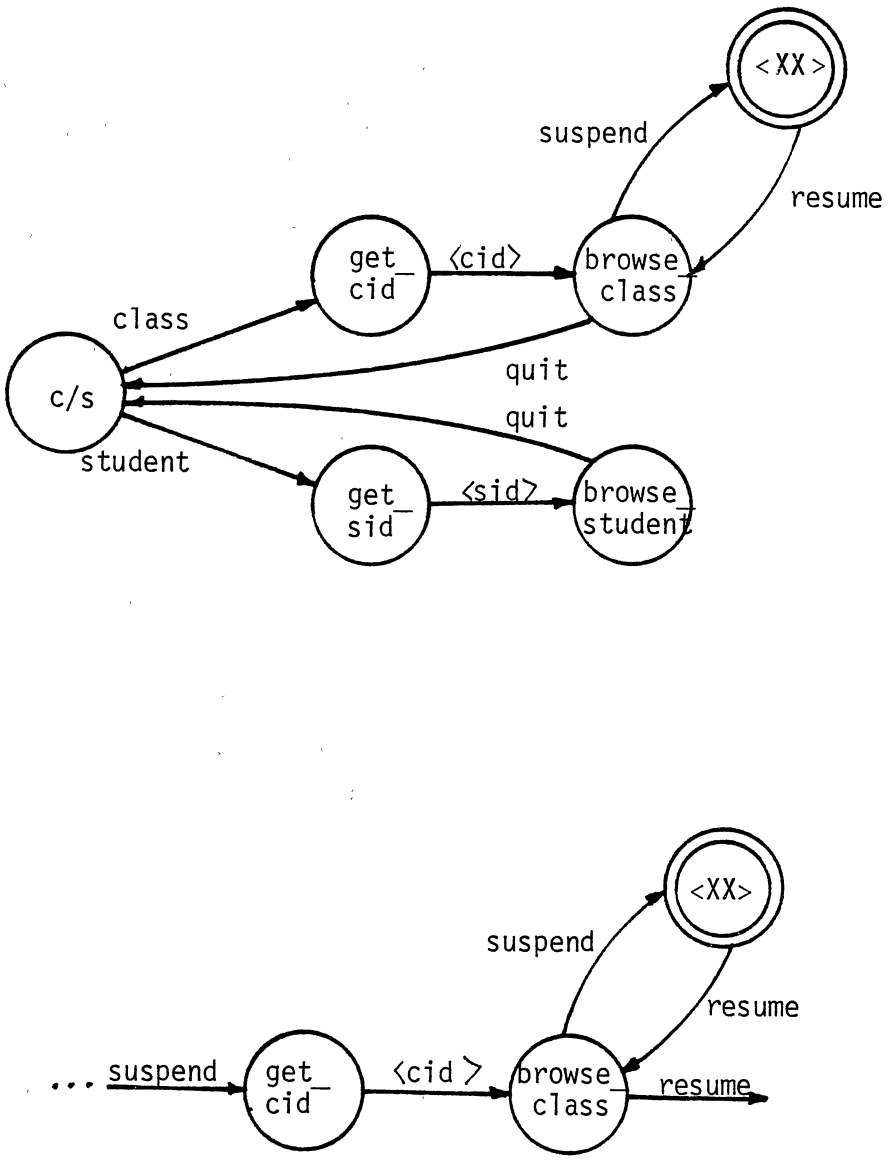

Figure 16. A Recursive Transition Network.

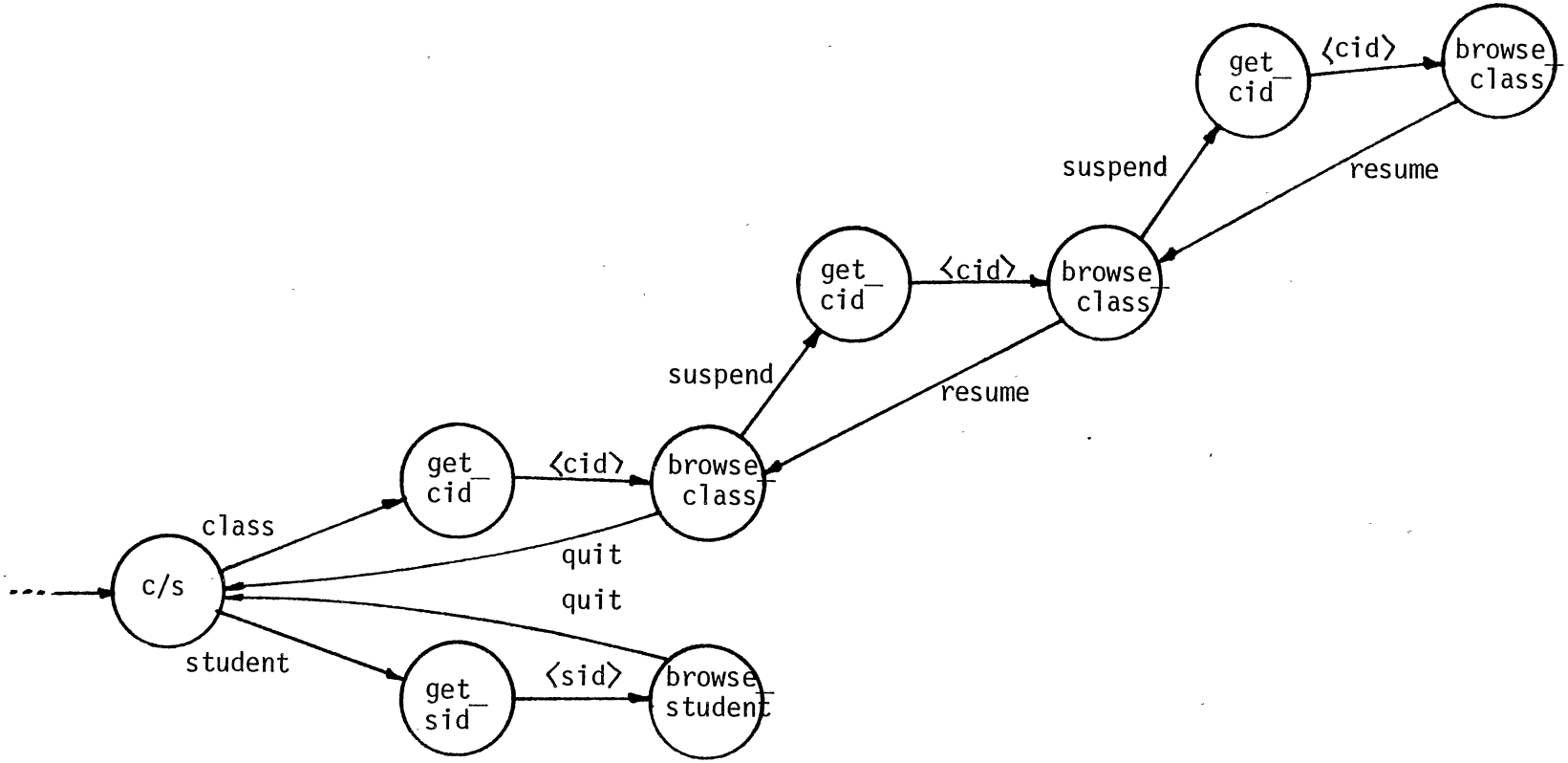

Figure 17. A Linearized Recursive Grammar.

the implementation of the finite state machine. The counter can be incremented by the actions that invoke the recursion and decremented with each level backed out of. The value of the counter can be tested to determine if the parser is at the top level or not. The use of a counter, however, only solves the problem of syntactic parsing. Semantically, if the environment is to be saved when a recursive rule is invoked, some type of stacking mechanism may still be required.

The Context-Free Grammar Model

The context-free grammar and its associated pushdown automaton recognizer describes a more powerful language than the language generated by a regular grammar. The ability to nest recursive calls to a rule indefinitely and pop out to the top level again is provided by the auxiliary memory the parser can use.

There is a graphical representation for a context-free grammar that is very much like the state transition diagram for regular grammars with complex states. If recursive calls to these complex states are allowed, the result is equivalent in power to a BNF grammar [6]. A common name for this graphical representation of a context-free grammar is a recursive transition network or RTN. An example RTN for the student grade keeping user interface is shown in Figure 16.

In order to gain explicit control over when an action is performed, a context-free grammar may have to be modified

(this is not the case for a regular grammar). This is because an action is associated with a rule reduction in a pushdown automaton parser. A finite state machine that recognizes a regular grammar executes an action with each token recognized.

With a user interface, it is particularly important to have explicit control over when actions are performed. Due to the interactive nature of the interface, many of the actions are prompts for the next input token. It would certainly limit the usefulness of the interface if several tokens had to be entered before the recognizer could execute the actions to prompt for them.

```
<S> ::= a<A>d      (1)
<A> ::= a<A>d      (2)
<A> ::= bc         (3)
```

Figure 18. A BNF Grammar

Figure 18 shows an example of a BNF grammar. The grammar recognizes strings of the form $\{a^i bcd^i | i>0\}$. If the action associated with the recognition of an "a" was to prompt for an "a" or a "b", it could not be executed until either rule (1) or rule (2) were reduced. Neither of these rules can be reduced until a "d" is recognized. By the time a "d" is recognized, there would be no need for a prompt to enter an "a" or a "b" because neither of these symbols could

follow a "d". In order to be sure an action can be executed immediately after recognizing the "a" in rule (1), an otherwise redundant rule that would be reduced when an "a" is recognized would have to be added to the grammar. The new nonterminal symbol associated with this new rule is then substituted for the "a" in rule (1). The resulting grammar is as shown in Figure 19.

```
<S> ::= <X><A>d
<A> ::= a<A>d
<A> ::= bc
<X> ::= a
```

Figure 19. A Modified BNF Grammar

This process may be carried out for each terminal symbol that appears with other symbols on the right hand side of any rule for which an action is required immediately after recognition.

Modifications of this type potentially add as many otherwise redundant rules to the grammar as there are terminal symbols. Indeed, the action associated with the "a" in rule (1) of Figure 18 may be different than the action associated with the "a" in rule (2). In this case, two separate rules are added to the grammar as shown in Figure 20.

```
<S> ::= <X1><A>d          (1)
<A> ::= <X2><A>d          (2)
<A> ::= bc                (3)
<X1> ::= a                (4)
<X2> ::= a                (5)
```

Figure 20. A Modified BNF Grammar

The action to be performed with the reduction of rule (4) would be different than the action associated with rule (5). These modifications may add as many rules to the grammar as there are terminal symbols on the right hand side of the productions that do not appear by themselves. Realistically, one would not expect the number of new rules to be quite that high. The addition of rules such as rules (4) and (5) in Figure 20 detract from the clean structure of the grammar and make it more difficult to read.

CHAPTER V

SUMMARY, CONCLUSIONS, AND SUGGESTED FURTHER RESEARCH

Summary

A user converses with an interactive software system by entering a series of discrete tokens, such as typed commands, keypresses, or joystick motions. The software system contains a user interface that accepts an input token. It determines if the token entered is a legal token in the context of the state of the software system and performs the action requested by the input token. The action may place a menu on the screen, prompt for more input, or perform a requested task.

An interactive session consists of a string of input tokens to the user interface, entered one at a time by a user, and a string of output messages generated as a response to the input tokens. The output messages may be produced by the user interface itself or by the underlying software system.

Typically, there are a finite number of distinct input tokens and an infinite number of ways they can be combined to form legal input strings to the user interface. However, not all sequences of input strings are legal inputs. A

formal grammar can aid in specifying the set of legal input strings.

A grammar uses a starting token and a set of production rules to generate all legal strings in a language. If the language is the set of all legal strings that a user interface will accept, the grammar that generates the language serves to specify the user interface. A recognizer based on this grammar specification can be constructed that will accept the legal strings in the language generated by the grammar. The recognizer constructed can be used as the driver for the user interface.

There are several advantages to specifying a user interface with a formal grammar. The recognizer built from the grammar specification can be the basis for an easily modifiable prototype that is quick to construct. Some of the well known advantages of rapid prototyping include the early detection of design flaws and early mock-up demonstration capability.

Another advantage of using a formal grammar to specify a user interface is the capability to automatically manipulate the grammar into an equivalent but more useful form. Well known algorithms to manipulate grammars include removal of useless production rules and removal of left recursion.

Other benefits center around the consistency and completeness of a formal model. Incompleteness and ambiguity show up more easily with a grammar model than if the interface was implemented in an ad hoc manner. The construction

of the grammar requires a complete design of the input language.

There are some drawbacks to using a formal grammar model for a specification of a user interface. One of the main drawbacks is that the grammar may be quite large if all legal strings are described. The application of semantic restrictions to a smaller set of production rules may help reduce the size of the grammar. This technique for reducing the size of the grammar is most effective for grammars that contain several rules which are nearly alike, with the only difference being a keyword. Other drawbacks relate to the time required to construct the grammar and the semantic problem of backing up the parser in the event of a dead end rule.

If a decision is made to use a formal grammar for the specification of a user interface, the type of grammar to use must be chosen. The most common types of grammars make up the Chomsky hierarchy of grammars. They are the regular, context-free, context-sensitive, and the unrestricted grammars. The focus of this study is on the regular and context-free grammar models.

The regular grammar model is the simplest model in the Chomsky hierarchy. It is recognized by a finite state machine which is compact, requiring no extra space for memory, and runs in linear time. The regular grammar is represented graphically by a state transition diagram. Actions to be performed by the recognizer are associated with a transition

from one state to the next in the state transition diagram. Each transition corresponds to a recognized token from the input. There is a great degree of explicit control of actions to be performed built into the regular grammar model.

A regular grammar may not be the best choice if the interface to be specified is very complex. User interfaces with recursive calls to rules are not representable by a pure regular grammar. There is no mechanism in a finite state machine to keep track of the number of levels of recursion the interface is nested. A regular grammar can still be used for this application if the rules are bent slightly and a counter is allowed.

Another solution is to use a context-free grammar and its associated pushdown automaton recognizer. The context-free grammar is a more powerful grammar for specification and the pushdown automaton contains a stack memory to aid parsing. The context-free grammar is typically represented in Backus Naur Form.

Actions to be executed are associated with a rule reduction in a context-free grammar. In order to gain explicit control over when actions are performed, the grammar may have to be modified. Rules that are otherwise redundant may have to be added to force the reduction of a rule when a particular token is recognized. These extra rules are only needed for semantic control purposes. They add size to what

may already be a large grammar and detract from the clean structure of the grammar.

Conclusions

The benefits of using a formal model over an ad hoc design are clear. A formal model provides a clear, consistent foundation upon which to build an effective implementation. The formal grammar seems to be a suitable model for the specification of user interfaces. The rigidity of the grammar model forces design decisions to be made at design time rather than during the implementation.

A particular strength of the formal grammar in conjunction with tools to automatically construct a parser is the capability for rapid prototyping. The grammar is easily modified if design flaws or awkward spots show up in the prototype. If a major modification is required, only the grammar needs to be rewritten. An ad hoc, hand-built prototype may have to be entirely rewritten.

When constructing the grammar, the designer should keep in mind that the grammar is specifying an interactive system. Difficulties can arise if any nondeterminism is introduced into the grammar. Backing up the parser in the event of a dead end rule may be difficult or impossible if actions that cannot be undone have already been performed. Moreover, in an interactive system the parser cannot look ahead to tokens that have not been input yet to determine an appropriate move.

The regular grammar model is the simpler of the two grammar models to implement. The built-in correspondence between the recognition of a token and the execution of an action makes this an appealing model. It should be powerful enough to specify a large class of useful user interfaces. This is especially true if the rules are relaxed a small amount to allow a variable for counting in the implementation.

The context-free grammar is the more powerful of the two grammars for specification. However, with the context-free grammar model, gaining explicit control over when actions are to be performed may cause the grammar to be cluttered by the addition of extra rules. However, it may be worth it for an interface that is very complex. If the rules for a regular grammar had to be bent too much, the regular grammar model would be cluttered also.

Suggested Further Research

During the investigation of grammars for the specification of user interfaces, it became clear that formal grammars might be useful for the specification of a more general class of software. Any system that requires the recognition of the input to perform actions seems to be a candidate for a formal grammar specification.

An even broader question is that of modeling software systems in general. The model could be a formal grammar or any other formal model that seems useful. The issue of

using a formal model to prove the correctness of a software system seems likely to provide fertile ground for research.

SELECTED BIBLIOGRAPHY

[1] Aho, Alfred V., and Ullman J. D. The Theory of Parsing, Translation, and Compiling, Vol. 1: Parsing, Prentice-Hall Publ. Co., Englewood Cliffs, N.J., 1972.

[2] Anson, E. The Device Model of Interaction. SIGGRAPH 82 (Computer Graphics), 16,3 (July, 1982), 107-114.

[3] Denert, Ernst. Specification and Design of Dialogue Systems with State Diagrams. Proceedings of the International Computing Symposium 1977, 4-7 (April, 1977), 417-424.

[4] Hanau, P. R., and Lenorovitz, D. R. Prototyping and Simulation for User/Computer Dialog Design. Computer Graphics, 14,3 (August, 1980), 271-278.

[5] Harrison, Michael A. Introduction to Formal Language Theory, Addison Wesley Publ. Co., Reading Mass., 1978.

[6] Jacob, Robert J. K. Using Formal Specifications in the Design of a Human-Computer Interface. Communications of the ACM, 26,4 (April, 1983), 259-264.

[7] Johnson, S. C., YACC: Yet Another Compiler Compiler. Unix Programmars Manual, Volume 2A (Janauary, 1979).

[8] Lawson, Harold W., M. Bertran, and J. Sangustin. The Formal Definition of Human/Machine Communications. Software - Practice and Experience, 8,1 (January, 1978), 51-88.

[9] Lesk, M. E., and E. Schmidt. LEX - A Lexical Analyzer Generator Unix Programmars Manual, Volume 2A (January, 1979).

[10] Olsen, Dan R., and Dempsey, Elizabeth P. SYNGRAPH: A Graphical User Interface Generator. Computer Graphics, 17,3 (July, 1983), 43-80.

[11] Pilote, Michael. A Programming Language Framework for Designing User Interfaces. ACM SIGPLAN Notices, 18,6 (June, 1983), 118-133.

[12] Reisner, Phyllis. Formal Grammar and Human Factors Design of an Interactive Graphics System. IEEE Transactions on Software Engineering, 7,2 (March, 1983), 229-240.

[13] UNIX is a Trademark of Bell Laboratories.

[14] Shaw, M., E. Borison, M. Horowitz, T. Lane, D. Nichols, and R. Pausch. Descartes: A Programming-Language Approach to Interactive Display Interfaces. ACM SIGPLAN Notices, 18,6 (Jun3, 1983), 100-111.

VITA

Michael Wayne Bates

Candidate for the Degree of

Master of Science

Thesis: FORMAL GRAMMAR SPECIFICATIONS OF USER INTERFACE PROCESSES

Major Field: Computer Science

Biographical:

Personal Data: Born in Ponca City, Oklahoma, October 21, 1959, the son of Kenneth and Carlene Bates.

Education: Graduated from Ponca City High School, Ponca City, Oklahoma, in May 1977. Attended Northern Oklahoma Junior College from August, 1977 to May, 1979. Received a Bachelor of Science degree in Computer Science from Oklahoma State University, July 1982. Completed the requirements for a Master of Science degree in Computer Science at Oklahoma State University, July 1984.

Professional Experience: Programmer, Department of Financial Aids, Oklahoma State University, May 1982 to October 1982; graduate teaching assistant, Department of Computer Science, Oklahoma State University, October, 1982 to December 1983; Programmer, TMS Inc., December 1983 to May 1984.