VALET: AN INTELLIGENT UNIX SHELL INTERFACE

by

Eric Norman Eide

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

The University of Utah

August 1995

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

Eric Norman Eide

This thesis has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

Chair: Robert R. Kessler

Thomas C. Henderson

Joseph L. Zachary

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the thesis of Eric Norman Eide in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Robert R. Kessler
Chair, Supervisory Committee

Approved for the Major Department

Thomas C. Henderson
Chair/Dean

Approved for the Graduate Council

Ann W. Hart
Dean of The Graduate School

ABSTRACT

Many modern human-computer interfaces are difficult for people to use. This is often because these interfaces make no significant attempt to communicate with the people who use them. In other words, these interfaces are uncooperative: They do not adapt themselves to their users' needs and they are insensitive to human foibles. Ordinary command line interfaces such as that of the UNIX C shell (`csh`) are intolerant of even the most simple input errors, even when those errors have obvious corrections. An "intelligent" UNIX shell interface, on the other hand, would make use of knowledge and interaction context in order to interpret — and as necessary, correct — its users' commands.

VALET is a prototype of such an "intelligent" interface to the UNIX C shell. VALET adds knowledge-based parsing and input correction to the shell by encapsulating an ordinary C shell process within a framework that allows VALET to control the shell's input and output. VALET intercepts shell commands and parses them, using its knowledge of the most popular UNIX shell commands, its built-in model of the file system, and data that describe the commands and files most often and recently referenced by individual users. VALET incorporates heuristics designed to detect and correct the kinds of mistakes that experienced users make most frequently: typographical errors, file location errors, and minor syntactic errors.

In order to evaluate the interface, eleven volunteers agreed to use VALET in the course of their normal work for approximately four weeks. The commands that those people entered, along with VALET's responses, were recorded and analyzed in order to measure the overall usefulness and effectiveness of the system. The data from the experiment suggest that knowledge-based, error-tolerant, "intelligent" command parsing can have very beneficial effects. The experiment also pointed to ways in which VALET could be improved.

To Shellie

CONTENTS

ABSTRACT . . . iv

LIST OF TABLES . . . viii

LIST OF FIGURES . . . ix

ACKNOWLEDGMENTS . . . x

CHAPTERS

1. INTRODUCTION . . . 1

- 1.1 The Need for Improvement . . . 2
- 1.2 The Benefits of Improvement . . . 7
- 1.3 An Intelligent UNIX Shell . . . 9
  - 1.3.1 Shortcomings of the C Shell . . . 10
  - 1.3.2 VALET: An Intelligent C Shell Interface . . . 13

2. PREVIOUS RESEARCH AND SYSTEMS . . . 17

- 2.1 The Potential for Improvement . . . 18
  - 2.1.1 Familiar Terms and Syntax . . . 18
  - 2.1.2 Flexible Parsing . . . 19
  - 2.1.3 Tolerance of Errors and Abbreviations . . . 20
  - 2.1.4 Use of Context . . . 26
- 2.2 The `metric` Library . . . 32
- 2.3 The `tcsh` and `zsh` Shells . . . 35
  - 2.3.1 Programmable Command Completion . . . 37
  - 2.3.2 Spelling Correction . . . 39
- 2.4 SAUCI, the Self-Adaptive User-Computer Interface . . . 43
  - 2.4.1 A Graphical User Interface . . . 44
  - 2.4.2 Adaptation to Individual Users . . . 44
  - 2.4.3 Context-Specific Advice . . . 45
  - 2.4.4 Orientation Toward High-Level Tasks . . . 46
  - 2.4.5 SAUCI Results . . . 47
- 2.5 SUSI, the Smart User System Interface . . . 50
  - 2.5.1 An Analysis of Users' Errors . . . 50
  - 2.5.2 The Design of SUSI . . . 53
  - 2.5.3 SUSI Results . . . 55

3. VALET . . . 58

- 3.1 The Goals and Limitations of the Interface . . . 59
  - 3.1.1 Intelligent Command Processing . . . 59
  - 3.1.2 Transparency . . . 60

3.1.3 Summary of Features and Limitations 63
3.2 An Overview of the Implementation 64
3.3 The GNU Emacs Components 68
3.3.1 Communication with the C Shells 70
3.3.2 Communication with Common Lisp 72
3.4 The Common Lisp Components 77
3.4.1 The Input Tokenizer 77
3.4.2 The Augmented Transition Network Parser 80
3.4.2.1 Transition Networks 81
3.4.2.2 Transition Network Actions 84
3.4.2.3 Explanation of Parsing Failures 90
3.4.2.4 Summary of the ATN Parser 92
3.4.3 Lexicons 93
3.4.4 The Shell Command Knowledge Base 95
3.4.5 The File System Knowledge Base 100
3.4.5.1 Representation of the File System 102
3.4.5.2 Examination of the File System 106
3.4.5.3 Parsing and Correcting File Names 106
3.4.6 Preservation of Interaction Contexts 111
3.5 The Process of Input Correction 112

4. EVALUATION 116

4.1 The Experiment 117
4.2 The Results of the Experiment 118
4.2.1 Analysis of All Accepted Inputs 123
4.2.2 Analysis of All Corrected Inputs 127
4.2.3 Analysis of All Rejected Inputs 129
4.2.4 Analysis of All Erroneous Inputs 132

5. CONCLUSION 141

REFERENCES 145

LIST OF TABLES

2.1 A Comparison of Two Editor Interfaces, Adapted from Ledgard et al. [19] . 18

2.2 Results of a Command Spelling Corrector, Adapted from Durham et al. [6] 23

2.3 Summary of Errors in Novice Users' Commands, Adapted from Jerrams-Smith [16] . . . 51

3.1 Summary of Messages Sent From GNU Emacs to Common Lisp . . . 73

3.2 Summary of Messages Sent From Common Lisp to GNU Emacs . . . 75

3.3 The Grammar Used to Define Transition Networks . . . 82

3.4 Summary of the `file-name` Parser Action Arguments . . . 107

4.1 Distribution of Correct and Incorrect Inputs Across VALET's Responses . . . 118

4.2 Summary of Users' Inputs and Errors . . . 120

4.3 Categorization of Accepted Correct Inputs . . . 124

4.4 Summary of the Most Frequently Invoked Commands . . . 126

4.5 Categorization of Accepted but Erroneous Inputs . . . 126

4.6 Categorization of Corrected Inputs . . . 128

4.7 Categorization of Rejected Inputs . . . 130

4.8 Categorization of Erroneous Inputs by Type . . . 133

4.9 Categorization of Erroneous Inputs by Outcome . . . 136

4.10 Categorization of Uncorrected Erroneous Inputs . . . 137

4.11 Categorization of Corrected Erroneous Inputs . . . 140

LIST OF FIGURES

2.1 An Annotated Transcript of `tcsh` Spelling Correction 40

3.1 Cooperating Processes Within VALET 65

3.2 Appearance of the VALET Interface Within GNU Emacs 69

3.3 Definition of the `token` Structure 78

3.4 Example Transition Network Definitions 85

3.5 Example Transition Network Action Definitions 88

3.6 Definition of the `command` Structure 96

3.7 Definitions of the `cd`, `cp`, and `rm` Commands 98

3.8 Definitions of the File System Model Structures 103

ACKNOWLEDGMENTS

I would like to thank Dr. Robert R. Kessler, the chair of my supervisory committee, for his encouragement, his insights, and his financial support for this work. Dr. Kessler gave me the freedom to research "intelligent" user interfaces even though my project was outside all of the established research areas within the University of Utah Department of Computer Science. Dr. Kessler provided equipment and funding for this research and also helped to test the intelligent shell interface during its infancy. For the freedom and support he gave to me, I will always be grateful.

I am also greatly indebted to the eleven volunteers who agreed to use VALET so that I could gather data about the interface. Without the help of these volunteers I would not have been able to evaluate the usefulness of the system.

In addition to the people who tested VALET, many other people contributed to this thesis as well. I am grateful for the assistance of Lal George, a research scientist at Bell Laboratories, who provided me with information about the `metric` library. I would also like to thank all of the people who proofread this thesis and made suggestions for improvement: my supervisory committee, Professors Robert R. Kessler, Joseph L. Zachary, and Thomas C. Henderson; my father, LeRoy N. Eide; and my wife, Shellie Eide.

I owe very special thanks to my wife, who never for a moment doubted that someday, somehow, I would finally finish this thesis. Her support made it possible.

CHAPTER 1

INTRODUCTION

Many modern computer systems are difficult for people to use. To a great extent, this is because these systems do not make any significant effort to communicate with the people who use them. In short, today's computer interfaces are uncooperative. They do not "speak" in human terms and they are not tolerant of human foibles.

A person using a computer quickly discovers that the machine has a language all its own — and that it insists on using exactly and only that language. Often, that language is a kind of textual *command language* that allows a human operator to communicate with a computer by typing commands on a keyboard. This kind of human-computer interface is referred to as a *command language interface* or *command line interface*. Unfortunately, many of today's command languages are poorly designed. They are built around confusing terms and cryptic syntax, and this causes computer users to make frequent input mistakes. Rather than help users to fix these mistakes, most command language interfaces instead compound the problem by being intolerant of even the smallest input errors — even those that have seemingly "obvious" corrections. These interfaces make absolutely no effort to determine what their human operators had intended to type. Instead, they simply report that the user has made an input error and then require the user to retype his or her entire command.

Because today's command language interfaces are usually so inflexible and intolerant of input slips, computer users quickly become frustrated or even intimidated by their computers' apparent unwillingness to communicate and "cooperate" with people. The result is that these computer systems, intended to be popular and powerful tools, are instead perceived as mysterious, unfriendly, and incomprehensible things.

In order to alleviate this situation, future computer systems must be designed to communicate more effectively with their human operators. Future command language interfaces will need to make greater efforts to understand the intentions of their users. In particular, these new interfaces will need to be tolerant of human errors.

This thesis describes an attempt to create one such interface of the future: VALET. VALET is an experimental "intelligent" user interface for the standard UNIX[1] C shell [17]. (The C shell is the program named "`/bin/csh`" on most UNIX systems.) The C shell is a program with a command line interface that allows its users to execute other UNIX programs. The C shell is therefore similar to the DOS command interpreter available on many personal computers. Although the C shell is a very complex program, it does not make any significant effort to understand its users' intentions. That is, the standard C shell takes input commands at "face value" and never attempts to correct its users' input mistakes. This behavior presents problems for both novice and experienced users.

VALET is an improved, "intelligent" interface to the C shell that addresses these problems. VALET is intelligent in the sense that it analyzes its users' commands, understands a more flexible command language, and is able to correct many common input errors including misspellings, typographical errors, and incorrect references to files. VALET incorporates a large body of knowledge about the UNIX system on which it runs, and in addition, it creates and maintains an individual interaction profile for each of its users. In this way VALET learns and adapts to the behavior of each of its human operators.

In summary, VALET goes to great lengths to understand the shell commands that are entered by its users. The result is that VALET is a more sophisticated, user-friendly, and cooperative command shell for UNIX.

1.1 The Need for Improvement

The study of *human-computer interaction* has grown rapidly in the past fifteen years along with the distribution of computers into businesses, schools, and homes. As the number of computer systems exploded, a new class of computer users emerged. These new users — including office staff, students, teachers, and writers — lacked the extensive training that had, until then, been a prerequisite for access to computer systems. These new "casual" users made clear a fact that had until then been largely ignored: Computer systems can be confusing and difficult to use.

Before the explosion in the availability of computers, computer scientists generally designed hardware and software systems without serious consideration for the ways in which people would interact with these products. Computer systems were created by

[1] UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

computer professionals for other computer professionals, or for other people with extensive computer training. Little thought needed to be paid to the design of a system's interface because both the designers and users of computer systems had similar technical backgrounds. In addition, the users of computers were much more interested in *what* their systems could do than they were in *how* those systems interacted with their human operators.

As computers became more and more commonplace, however, the emerging class of "casual" computer users had very different requirements. Although the computational features of a system were still very important, it was now equally important that these features be made easily available and understandable to the operators of the system. In other words, computer interfaces now had to be *user-friendly.* People with little or no computer training now needed to use computers as writing tools, filing tools, computational tools, and data access tools — all without being required to understand much if any of the science behind the computer systems.

It quickly became apparent that most systems did not have the user-friendly interface that was required by the new casual users. In 1985 Bertino wrote [2]:

> The recent proliferation of computer equipment has not been accompanied by a comparable increase in user-friendly interfaces. Users of advanced hardware machines are often disappointed by the cumbersome data entry procedures, obscure error messages, intolerant error handling, and confusing sequences of cluttered screens. In particular, novice users feel frustrated, insecure or even frightened when they have to deal with a system whose behavior is incomprehensible, mysterious, and intimidating.

Because many human-computer interfaces were confusing and frustrating, the effectiveness of many computer systems was substantially reduced. It became clear that to a large degree, the usefulness of a computer system was limited by the effectiveness of the system's interface. Particularly for novice computer users, *what* a system could do was determined by *how* the system communicated. Software and hardware designers began to realize that they had failed to provide effective user interfaces — not just for novices, but also for themselves and other technical users. The UNIX operating system, for example, was and still is a very popular operating system for the powerful computers used by scientists, engineers, and other technically literate people. One might assume that UNIX computer systems would have the most effective user interfaces that computer scientists could design, but that assumption would be far from the truth. In fact, UNIX systems to this day have a well-deserved reputation for being among the *most difficult* and confusing systems to use. Even technical professionals find UNIX to be confusing and frustrating. Norman [25, page 139], for example, wrote that the UNIX operating system's command shell was a user interface "disaster." "It fails both on the scientific principles of human engineering and even in just plain common sense."

Because clumsy user interfaces were an obvious impediment to the widespread usability of computers, the study of human-computer interaction grew quickly. Today, the design of human-computer interfaces is a major research and commercial concern. It is standard practice for software companies to evaluate and refine their products' interfaces through user-testing experiments in which people are asked to perform various tasks with the software.

Since the discovery of the importance of interface design, command line interfaces like the UNIX shell have been widely criticized by researchers as "unfriendly" for a number of reasons:

Confusing terms. Command languages sometimes contain terms that are meaningful to the people who *design* the language but that are not meaningful to the people who *use* the language. For example, there is a common UNIX program that prints a message every time a new electronic mail message arrives for a user. To run this program, a person must type the program's name: `biff`. The name `biff` is entirely meaningless to most UNIX users and few novices would guess that `biff` has anything to do with electronic mail. (According to folklore, the program was named after a dog that barked whenever a postal carrier arrived.) Many other UNIX program names are abbreviations or acronyms — `mv`, `cp`, `rm`, and `grep` — and although short names can reduce the need for typing, commands like these ultimately make the UNIX shell language inaccessible to casual users. Even experienced users can forget or stumble over poorly chosen terms.

Inflexibility. Command line interfaces are inflexible in the sense that there is often only one way to phrase a particular command. Commonly, a command language interface will recognize only one name for each available action, object, or other entity — no synonyms, abbreviations, or misspellings of the names will be tolerated. Many languages are based upon strict syntax rules that do not allow the user to phrase commands in all of the ways that seem natural — for instance, it may not be possible to specify command options in an arbitrary order. While restrictions such as this may be acceptable to expert users of a system, inflexibility frustrates people who are not so familiar with the language. Inflexible systems are perceived by novice users as intolerant and uncooperative, and sometimes even as intimidating. Not surprisingly, researchers have shown that flexible command language interfaces are easier and more enjoyable for novices to use [9].

Inconsistency. Command languages often contain inconsistencies. For example, some commands may be abbreviations of English words whereas others may be word truncations or phrase acronyms. Some commands in a language may accept arguments or options that other similar commands do not. Even worse, the syntax of a language may seem to differ between commands. The various UNIX command shells have been particularly criticized in this regard [25]. (It is important to note, however, that most of the UNIX shell's language is actually interpreted by programs other than the shell itself. As described in Section 1.3.1, the shell language is inconsistent largely because these other UNIX programs are inconsistent with each other.)

Lack of feedback. In many situations, command line interfaces provide little or no response to their human operators. Many interfaces are designed around the principle that silence is desirable: Informational messages should be displayed only in response to specific requests for data or in order to inform the user about exceptional conditions (e.g., errors). Although this principle can greatly reduce the amount of unwanted "noise" in an interface, it can also make casual users uncomfortable because there is no apparent indication that the computer is doing anything in response to users' commands. Furthermore, a lack of informational responses tends to put a burden on users' memories. Users can quickly forget what state a system is in. This was demonstrated, for example, in a study of UNIX shell command use by Hanson et al. [12] who found that 21% of all users' shell commands were orienting commands that describe the current environment (e.g., the contents of the current directory).

One of the most popular ways to handle these command line interface problems has been to abandon command line interfaces altogether. In recent years many computer applications have adopted *graphical user interfaces*, which are often believed to be easier for casual users to master. Graphical interfaces usually address all of the problems listed above; for example, graphical interfaces tend to display more orienting information (e.g., the set of available commands) than command line interfaces do. It is difficult, however, to conclude that graphical user interfaces solve the above problems by their very nature. It is possible to imagine graphical interfaces that provide minimal user feedback, that behave in inconsistent ways, that use confusing pictures and words, and so on. Rather than conclude that graphical interfaces are inherently superior to command line interfaces, it is more reasonable to believe that the designers of graphical interfaces have simply learned from the mistakes made in older command line interfaces.

Graphical user interfaces are certainly appropriate for many applications, but they are not the ideal interfaces in all situations. Command language interfaces are widespread on today's computer systems and are effectively used in a variety of computer applications — including command shells, electronic mail programs, debuggers, and information browsers — on machines ranging from personal computers to supercomputers. Although graphical user interfaces are becoming increasingly popular, command line interfaces have unique and important strengths:

1. For many tasks, command language interfaces are considered to be more expressive than graphical interfaces. Compared to today's graphical user interfaces, command line interfaces often provide more powerful ways for users to enter commands. This is because command line interfaces provide syntax for:

   (a) specifying details and command options in a succinct manner;

   (b) referring to groups of similarly named objects through the use of patterns;

   (c) combining two or more commands into one larger command unit; and

   (d) recalling and reexecuting previously entered commands.

   It is often possible for a person to carry out a complex task with only a few keystrokes to a command line interface when it would require significantly more effort for that person to perform the same task by "pointing and clicking" with a mouse in a graphical interface. In general, the disparity between the two interface methods grows as the task becomes more complicated and detailed.

2. Because command line interfaces are textual, it is straightforward for a person to create computer files that contain sequences of commands for such interfaces. These files are called *scripts* or *batch files* and are used to simplify and automate complex tasks. Once written, a script can easily be executed many times. Furthermore, if the command language is sufficiently powerful, scripts and batch files can behave like full-fledged programs. They can make decisions and adapt to changing situations just as ordinary programs do.

   Most systems that provide command line interfaces are also able to process batch files. Current graphical user interfaces, however, generally do not provide similar facilities. At best, graphical interfaces allow users to record and play back sets of graphical gestures. Although these recording facilities can be useful, they cannot offer the kinds of programming features found in many command line interfaces. A recorded series of gestures cannot make decisions or adapt to the current state of the system, for example.

   The need for script and batch file programming features is demonstrated by computer applications that provide both a graphical interface and a separate scripting language. In these systems, the scripting language is essentially a noninteractive command language interface to the program.

3. Finally, existing operating systems (e.g., UNIX and DOS) provide command line interfaces, and these operating systems will continue to be popular for many years.

For all of the above reasons, command language interfaces will be important to many computer systems of the future. Therefore it is necessary to identify the problems in today's command line interfaces and determine how those interfaces can be improved in order to meet the needs of their users more effectively. Chapter 2 describes several attempts to explore and design user-friendly, error-tolerant, and "intelligent" interfaces, including several efforts to improve the interface of the UNIX shell in particular.

1.2 The Benefits of Improvement

As Card observed [3], when a person uses a computer system with an intelligent interface, the interaction is less like the use of a tool and more like participation in a conversation. A tool is generally a passive object; when a person uses a tool, the person is active and the tool is passive. A tool takes no initiative and makes no attempt to understand its user's intentions. Compare this situation to that in a conversation. In a conversation, all of the participants are active. Initiative is shared; at different times, different participants may direct the interaction. Responsibility for successful communication is also shared. The agents in a conversation pay attention to each other and attempt to understand each other in order to communicate or accomplish a task — in other words, the agents cooperate.

A primitive human-computer interface forces its user to adapt to the computer; the computer makes no effort to adapt to its user. Because the interface is fixed and passive there is no cooperation. The human operator must take all of the initiative in the dialogue, and responsibility for successful communication with the computer system is entirely up to the user. This imposes a great burden on the user. When combined with confusing command languages, obscure syntax, and lack of feedback, these passive, tool-like qualities are what cause ordinary computer interfaces to appear uncooperative, unfriendly, and even hostile.

Intelligent interfaces, on the other hand, are better able to cooperate and communicate with their human operators. An intelligent command line interface makes use of domain-specific knowledge and contextual information which allows it to:

- accurately correct many errors in users' input commands;
- accurately interpret user-chosen abbreviations;
- infer users' intentions and goals;
- predict users' actions;
- offer context-specific assistance or advice; and
- adapt to the experience levels and idiosyncrasies of individual users.

In combination with user-centered design techniques — the use of familiar terms and paradigms in the command language, flexible parsing, and appropriate feedback for users' actions — these cooperative, conversation-like abilities are what cause intelligent interfaces to appear user-friendly, helpful, unintimidating, and accessible. Both novice and experienced computer users can benefit from these kinds of improvements in human-computer interfaces. Because intelligent interfaces remove many of the barriers that frustrate users, novices will be more willing to use computers, and people at all levels of experience will be more satisfied and able to use computer systems more productively.

The intelligent features listed above are clearly useful in interactive command language interfaces such as those described in Chapter 2. In addition, these features can be useful in other kinds of applications as well. For example, error tolerance and flexible parsing can be incorporated into batch systems such as compilers and scripting languages. Morgan [23] did this in 1970 and it was very useful then, so it is surprising that typical modern software engineering environments do not offer this kind of intelligence today. Intelligent behavior can also be incorporated into graphical user interfaces and vocal interfaces. Graphical interfaces have many textual components such as "search and replace" dialogs that could benefit from intelligent text processing. Vocal interfaces will clearly need to incorporate conversational qualities in order to understand spoken input, which is often naturally fragmentary. People are so accustomed to speaking in conversational modes that in order to be most useful, vocal interfaces will be required to use contextual information to understand spoken input. Efforts to create intelligent, conversational command line interfaces today, therefore, will also lead to improvements in tomorrow's multimedia interfaces.

In summary, if computers are going to become as widely useful as possible, then human-computer interfaces must be improved. Future command language interfaces will need to be conversational and will need to make substantial efforts to understand the intentions of their users. Today's interfaces, such as that of the UNIX shell, will need to be improved.

1.3 An Intelligent UNIX Shell

A *UNIX system* is a computer that runs a particular kind of operating system and which offers a certain set of standard utility programs. This set of programs always includes one or more *shells* (also called *command shells*). Essentially, a shell is a special program that allows its user to invoke and coordinate the other programs and resources that a computer system provides. A shell is therefore a crucial component of almost all interactive computer systems — without some kind of command shell, it is impossible for a person to start new programs.

Because of the shell's special role, an interactive computer system usually starts a shell automatically whenever a person begins a computer session. For example, when a person logs into a UNIX system, a new shell process is automatically created for that user. The shell takes control of the user's terminal and prompts the user for input. By entering commands to this shell the computer user can run the other programs that the UNIX system offers — text editors, compilers, debuggers, typesetters, and so on. Those other programs may temporarily take control of the user's terminal, but when those programs terminate, the shell reclaims control of the terminal and prompts the user for additional commands. Because of this behavior, computer users often perceive the shell to be the fundamental interface of the computer system itself. The shell appears automatically at login, allows users to run the system's programs, and reappears when those other programs terminate; the shell, therefore, largely defines the personality of the computer system as a whole. If a computer's shell is confusing and difficult to use, then users will generally believe that all of the system is confusing and difficult to use. It is evident, therefore, that in order for a computer system to be perceived as intelligent and user-friendly, it is essential that the computer system's command shell demonstrate these same qualities.

1.3.1 Shortcomings of the C Shell

Unfortunately for users of UNIX systems, the most commonly used UNIX command shells are neither intelligent nor user-friendly. One of the most popular UNIX shells is the C shell [17] — the program named "`/bin/csh`" on most UNIX systems — which implements a command line interface with an expressive command language.[2] However, although the C shell is a complex and powerful program, it makes practically no effort to understand the intentions of its users. It accepts all inputs at "face value" and never attempts to correct its users' input mistakes. This uncooperative, unhelpful behavior frustrates users at all levels of experience, and many computer scientists have complained that traditional UNIX shells demonstrate a complete disregard for effective user interface design principles. Gabriel and Steele, for example, have described how frustrating a typical UNIX shell can be [8]:

> Computers have no idea what is going on. You can't hold a reasonable conversation with them, even on their own terms. Does the following scenario look familiar?

[2]While there are several popular command line shells for UNIX, the standard C shell is both typical and among the most widely used.

```
% lpt /usr/fred/common-lisp-functions
lpt: Command not found.

% lpr                                          <long pause>
/usr/fred/common-lisp-functions                <another pause>
^C

% lpr /usr/fred/common-lisp-fucntions
lpr: cannot access /usr/fred/common-lisp-fucntions

% lpr /usr/fred/comon-lisp-functions
lpr: cannot access /usr/fred/comon-lisp-functions

% lpr /usr/fred/common-lisp-functions          <typed slowly
                                                and with care>
lpr: cannot access /usr/fred/common-lisp-functions

% ls common*

% ls /usr/fred/common*
/usr/fred/common-lisp-fns

% lpr common-lisp-fns
lpr: cannot access common-lisp-fns

% /usr/fred/common-lisp-fns
/usr/fred/common-lisp-fns: Permission denied.

% lpr /usr/fred/common-lisp-fns

%                                          <success at last>
```

In the preceding transcript, the computer user made one small mistake after another — first misspelling the command `lpr`, then forgetting to retype the file name in the corrected command, then mistyping the file name, and so on. The shell itself was no help; it did nothing to correct these mistakes, nor did it attempt to make plausible interpretations of the erroneous commands. By the time the user finally entered the command that he or she originally intended, the user must have been extremely frustrated with the shell. The above transcript, although unusually extended, truthfully illustrates the kinds of mistakes that C shell users make every day and the C shell's typical responses to those mistakes. In short, the C shell can be very uncooperative — to novice users, even antagonistic.

The C shell also suffers from the other command line interface problems described in Section 1.1. The input language of the shell is confusing, especially for inexperienced users, and there is often little or no feedback in response to users' input commands. These problems, however, are largely not caused by the C shell itself but are instead caused by the other programs that make up a UNIX computer system. Because the purpose of the C shell is to invoke other programs, the input language of the shell is largely made up of the names of those other programs. Many of those programs accept arguments — words that specify or control the behavior of a program — and so the typical command to the C shell has this form: "*program-name argument-1 argument-2* . . . ." When a C shell user wants to invoke a program, he or she types the name of that program, followed by zero or more arguments, as a single command to the shell. The shell then starts the named program, giving that program the set of arguments that the user typed on the command line. This is a natural and obvious way for users to invoke UNIX programs, but the result is that the C shell effectively does not define its own command language. The set of C shell commands is largely the set of names of the *other* programs that are available, and the C shell cannot control that set. Furthermore, each program can interpret command line arguments in any way it wishes. The C shell does not know what arguments any other program may expect or how those arguments might be interpreted. Each UNIX program imposes its own syntax rules on its command line arguments, and although there are a few general syntax rules that most UNIX programs use (e.g., that an option begins with the character "`-`"), many programs apply these rules inconsistently or ignore them altogether.

Not surprisingly, the ad hoc design of the C shell's command language causes the shell to be confusing and hard to use. The names of common commands — such as `grep`, `biff`, and `awk` — were chosen to be meaningful or clever to the authors of those programs. Today, however, these program names are nonsensical to most UNIX system users and this makes the C shell language seem ridiculous and arbitrary. The conflicting rules for different programs' command line arguments add to the confusion. Users must remember the individual command line idiosyncrasies of many different commands, which puts a large burden on users' memories and which causes users to be frustrated when they forget. Finally, many common UNIX programs are designed to print messages only when it is absolutely necessary to do so (e.g., in response to a specific request for information) or when an exceptional circumstance arises (e.g., when an error occurs). Many of the most frequently used UNIX utilities print no messages at all in normal operation. Although this "silence is golden" behavior is sometimes convenient for sophisticated UNIX users, it often leaves novice UNIX users wondering if the commands that they give to the shell are succeeding — or are even being accepted at all! These silent programs make it appear that the C shell is unwilling to share information with the people who use it. In this way, the silence of other UNIX programs reflects badly on the shell itself.

The shell in turn reflects on the computer system as a whole. As previously described, because the shell is such an integral part of the computer system, the shell has a great influence on its users' perceptions of the computer system: If the shell is unfriendly, the entire system appears to be unfriendly. It is not hard, then, to see why UNIX systems have a widely known reputation for being difficult and confusing to use. UNIX systems are intimidating because their command shells are confusing. The C shell input language is confusing and inconsistent, and the shell makes no attempt to correct its users' input errors. Because many UNIX programs are usually silent, the command shell itself appears to be unwilling to share information. The shell does not meet the needs of its human users, both novice and expert, as effectively as it should, and the result is that UNIX systems have a well-deserved reputation for being impenetrable.

Norman [25, page 139] wrote that the UNIX shell interface was a user interface "disaster" that needs to be corrected: "If UNIX is really to become a general system, then it has got to be fixed. I urge correction to make the elegance of the system design be reflected as friendliness toward the user, especially the casual user." The place to start is with the UNIX command shell.

1.3.2 VALET: An Intelligent C Shell Interface

In order for UNIX systems to best meet the needs of their users, the UNIX command shell must be changed. It must do more than simply accept command lines from people and pass that input, uninterpreted, to other UNIX programs. Rather, the shell must actively assist its users. The shell must attempt to understand the intentions behind its users' input commands and it must help its users correct input errors — in other words, the shell must behave "intelligently." The VALET interface to the UNIX C shell is a step in this direction.

VALET is an experimental intelligent interface to the standard UNIX C shell. Through the use of context as described in Section 2.1.4, VALET attempts to understand its users' commands. VALET uses knowledge of the UNIX system on which it runs, combined with data from individual users' sessions, in order to analyze input commands. This means that VALET can automatically and accurately correct many common input errors including misspellings, typographical errors, and incorrect references to files. VALET maintains a separate interaction profile for each of its users, so it is able to adapt itself to the habits of each of its human operators.

Although VALET uses knowledge in order to interpret commands, VALET does not fundamentally change the ordinary input language of the C shell. Instead, VALET applies intelligent processing to the existing language. As previously described, rather than being defined by the C shell itself, the language of the shell is the combined product of the hundreds of other UNIX programs available on the computer system, and each of those programs has its own unique quirks. Even though the language of the C shell is far from optimal, it would be a massive undertaking to replace the shell's language. One would either need to change hundreds of existing programs and insure that future programs adhere to the new interaction guidelines, or one would need to rewrite the shell so that it offers a new language to its human users and translates that language into the forms expected by other UNIX programs. Each of these alternatives is untenable. The first would require one to change long-established and standardized programs and would force one to rewrite all existing shell scripts and other similar software — a truly monumental task! The second alternative, changing the shell's interaction language, would confuse users because the shell would disguise the native interface to other programs. Because the shell is not the sole interface to other UNIX programs, it is likely that many users (especially sophisticated users) would have to know both the new shell language and the native language of other programs. The shell language would have to be frequently updated as new programs were added to the system. Moreover, a new shell language would be objectionable to people who are already comfortable with the existing UNIX shell interface. For all of these reasons, VALET attempts to interpret rather than replace the current language of the C shell.

Similarly, VALET does not attempt to change the user interface of any program except the C shell. VALET does not, for example, attempt to analyze or clarify error messages from other UNIX programs. Although it might be useful for a shell to do this — in order to adapt error messages to an individual user's level of experience, or in order for the intelligent shell to establish additional context, for example — this task would require extensive changes to hundreds of pieces of UNIX software. VALET, therefore, is restricted to the task of analyzing, understanding, and correcting users' inputs to the C shell.

Although the addition of intelligence to the C shell is clearly important, the shell's language has many features that are not conducive to intelligent parsing:

- Most significantly, most of the shell's language is implemented by other UNIX programs as previously explained. This means that rather than being fixed, the language changes as programs are modified or added to the UNIX system. Normally the shell has no knowledge of other programs' interfaces (i.e., the required syntax of their command line arguments), but an intelligent interface obviously needs this information. VALET, therefore, has a knowledge base that describes the interfaces of the most commonly used UNIX programs. The knowledge base cannot describe every available program, however, so VALET must also deal with programs with unknown interfaces. (Not surprisingly, VALET is less intelligent when dealing with these unknown interfaces.)

- The set of available commands and their interfaces are not the only things that VALET cannot control but must still understand. Many programs expect to receive file names as arguments, for example, so VALET must maintain its own representation of the file system. This is complicated by the fact that the file system changes frequently and that it is very time consuming to scan the file system.

- The shell language is full of features — aliases, file name patterns (called *globs*), variables, pipelines, and input and output redirection — that can be difficult for an intelligent interface to handle properly. VALET in fact understands only a subset of these features.

- Finally, the shell language is generally terse, so it can be difficult for an intelligent parser to rely on lexical information. This fact is illustrated by the "`dc`" and "`cd`" shell commands. The command "`dc`" is valid (`dc` is a standard UNIX program that simulates a desk calculator) but most users never invoke `dc`. On the other hand, the "`cd`" command, which changes the shell's current directory, is invoked frequently by all shell users. An intelligent interface, therefore, must make a decision when a user enters the command "`dc`": Did the user intend to invoke `dc`, or did the user made a mistake while typing "`cd`"? Because these two commands have such similar, short names, lexical context alone is insufficient for an intelligent shell to make an accurate determination. For this reason, VALET maintains and refers to other kinds of context in order to make decisions such as the one described above.

By making use of many kinds of knowledge, VALET attempts to overcome the above-described problems with intelligent parsing of shell commands. VALET is therefore able to understand most of its users' inputs and can accurately correct the most frequent types of input error. VALET is not always able to discern its users' intentions, of course, but it can almost always make reasonable interpretations within the limitations of its knowledge bases. VALET is unobtrusive and differs from the standard C shell interface only when it detects an input error. Overall, because VALET can accurately correct the most common user input errors, it makes a significant contribution to the effectiveness and friendliness of the UNIX C shell interface.

Chapter 2 describes research that influenced the design of VALET. Readers with limited time may wish to skip Chapter 2 and read just the following portions of this thesis: Section 3.1.3, summarizing the goals and limitations of the interface; Section 3.2, describing the overall implementation; and Chapter 4, pages 116 through 123, explaining the user testing experiment and summarizing the results.

CHAPTER 2

PREVIOUS RESEARCH AND SYSTEMS

VALET is not the first attempt to create a more intelligent, more accommodating command shell for the UNIX operating system. Prior to the design and implementation of VALET, several other systems demonstrated that it was both possible and useful to improve the UNIX shell. VALET was influenced by the systems that preceded it, and in turn, those systems were influenced by the results of even more previous user interface research. As described in Section 1.1, the need for improved human-computer interfaces became clear within the past ten or fifteen years as computer systems became more widespread and typical computer users became less technically oriented. Both novice and experienced computer users began to demand that more attention be paid to the design of effective user interfaces. It is not surprising, therefore, that as the UNIX operating system became increasingly popular, its user interface deficiencies became both more apparent and more serious, and computer scientists turned their attention toward addressing the UNIX command shell's shortcomings.

This chapter describes some of the user interface research that influenced VALET. In particular, several attempts to improve the UNIX shell interface are presented. Some of these efforts are research vehicles that explore how certain techniques such as spelling correction or task-specific knowledge can be employed to improve the UNIX shell interface. Other systems described in this chapter — the `tcsh` and `zsh` command shells in particular — are in actual widespread use today as replacements for the aging UNIX C shell. These new UNIX command shells incorporate many improvements over the standard C shell. VALET incorporates many of the interface improvements offered by those shells and in addition builds on those ideas by maintaining extensive user interaction contexts. In this way VALET goes beyond the capabilities of today's popular UNIX shells.

2.1 The Potential for Improvement

As described in Section 1.1, many command line interfaces can be difficult for people to use. Confusing terms, inflexible syntax, inconsistency, and lack of feedback can all reduce the effectiveness of an interface. Although with training people can adapt to the requirements of almost any computer system, it is clearly preferable for computer systems to adapt to the requirements of their users. People at all levels of experience can benefit from user-friendly interfaces. Fortunately, research with existing command language interfaces has shown that there are many ways in which these interfaces can be improved.

2.1.1 Familiar Terms and Syntax

Confusing, hard-to-remember terms in a command language can be replaced with words that are more familiar to the people who use the interface. Similarly, the syntax of the language can be changed to be more natural to those users. Several groups of researchers have shown that these modifications can greatly improve a command line interface, as evidenced by increased user performance and satisfaction with the system. Ledgard et al. [19], for example, compared two interfaces for a text editor: a traditional "notational" interface and a "natural language" interface based on English words and phrases. Twenty-four paid volunteers with various levels of computer experience were asked to perform a set of editing tasks with both editor interfaces. Half of the subjects used the notational editor first; the other half used the English editor first.

The results of this experiment are summarized in Table 2.1. Ledgard and his colleagues discovered that overall, the test subjects were more productive with the English-based editor interface. The subjects completed an average of 48% of the editing tasks with the

Table 2.1. A Comparison of Two Editor Interfaces, Adapted from Ledgard et al. [19]

Subjects' Level of Experience	**Mean Percentage of Tasks Completed**		**Mean Percentage of Erroneous Commands**	
	Notational	**English-based**	**Notational**	**English-based**
Inexperienced	28	42	19	11
Familiar	43	63	18	6.4
Experienced	74	84	9.9	5.6
Average	48	63	16	7.8

notational editor, but they completed an average of 63% of the tasks with the English-based editor. Users at all levels of experience showed significant improvement. (Ledgard et al. noted, however, that the users tended to improve with exposure to the task and therefore tended to be more effective with whichever editor they used second.) The test subjects entered fewer erroneous commands to the English-based editor. In addition, at the end of the experiment, users at all experience levels clearly preferred the English editor. Ledgard and his colleagues concluded that command languages based on everyday speech can lead to increased user efficiency and satisfaction.

2.1.2 Flexible Parsing

Another way to increase user performance is to make command languages more flexible. As previously described, many of today's command languages recognize only one name for each available action, object, or concept in the system. It is straightforward to correct this situation: Interfaces can be designed or changed to accept synonyms. In addition, command language syntaxes can also be expanded to accept a wider variety of command phrasings. These simple changes can have a very powerful effect, as Good et al. [9] discovered.

In their study, Good et al. set out to determine how a particular command line interface could be modified to accept commands that "seemed reasonable" to inexperienced computer users. At the beginning of the experiment, the researchers created a command line interface to a simulated electronic mail system. This original interface was similar to those of several actual electronic mail systems. The researchers then recruited novice users, none of whom had any experience with electronic mail, to perform a set of tasks with the mock system. Each test subject received a brief introduction to the concepts of electronic mail and a general description of the command line interface; however, the subjects were not told what commands or syntax the program understood. After each user's orientation, the researchers left the user to accomplish the electronic mail tasks without human guidance or online assistance.

The user testing sessions were divided into several groups. After each round of tests, Good and his colleagues analyzed the logs from each session and updated their mock computer mail system accordingly. The researchers enhanced the system's command parser to allow it to recognize a greater percentage of the test subjects' commands. These enhancements were guided by the commands actually attempted by users; in this way, the interface was derived from user behavior.

As the system's parser became more sophisticated and flexible, it recognized a much greater percentage of the commands spontaneously entered by the test subjects. Of the 1,070 commands entered during the entire course of the study, only 78 (7%) could be handled by the initial version of the parser. The final version of the parser, however, incorporated 30 changes and could correctly recognize 816 (76%) of the commands. The effectiveness of the parser increased by an order of magnitude, and the final parser could understand over three-fourths of the novice users' commands. There was an additional effect from the improvements in the parser: Users increasingly enjoyed working with the system. The test subjects were neutral toward the initial version of the interface, but they liked the final version. Eight experienced computer users were asked to use the final system and they liked it as well.

Good and his colleagues concluded that the examination of novice user behavior is an effective technique for creating natural, easy-to-use computer systems. In addition, the researchers concluded that flexibility is a very important aspect of effective command line interfaces. Good offered the following guidelines:

1. Command language interfaces should be designed to accept synonyms. Good estimated that if a computer system knows only one word for an object or concept, then there is an 80% to 90% chance that a user will fail to guess the computer's word. In the case of the electronic mail system, Good et al. wrote [9, page 1,038]: "The most effective change [to the interface] permitted the recognition of the three most widely used synonyms for commands and terms. One-third of all commands issued required this change for successful parsing."

2. Command language interfaces should allow objects to be described in flexible ways. Numbers, for example, might be preceded by "#". Furthermore, it should be possible to reference objects through their attributes. In the case of Good's final mail system, users could refer to messages by any header field: subject, author, date, or other.

2.1.3 Tolerance of Errors and Abbreviations

Good et al. improved their program's interface by expanding its command parser to recognize the variety of terms and syntaxes that were used by the test subjects in their study. This clearly made their interface much more flexible, but by itself, the addition of new terms to a language enables only a "static" kind of flexibility.[1] While a large vocabulary may allow an interface to recognize many different words, it does not necessarily allow an interface to recognize *variations* or *approximations* of those words. Even if a command line interface is designed to understand synonyms and a flexible command syntax, in order to be maximally flexible the interface must do more than accept only and exactly the preprogrammed terms. It must also attempt to handle dynamically chosen approximations of those terms, including abbreviations, misspellings and typographical errors.

It is hardly a new idea that computer systems should be tolerant of users' command input errors and of user-chosen abbreviations. Over thirty years ago, Damerau [5] described a technique for correcting input errors in an indexed information retrieval system. This system compared input index keywords against a master list and rejected those keywords that were not in the list. Examining these rejected terms, Damerau discovered that a great majority of the input errors were exceedingly simple. Over 80% of the unrecognized input terms were simple lexical transformations of terms that were known to the program.[2] In particular, Damerau discovered that each of these erroneous input terms could be classified as the result of exactly one of the following typographical mistakes:

Omitting one character. The erroneous input term could be produced by deleting one character from some term in the program's lexicon. For example, the computer operator may have entered the term "`comuter`" when he or she had intended to enter the (known) term "`computer`".

Inserting one character. The erroneous input term was the product of inserting one character into a known term. For instance, the user may have entered the term "`commputer`" when "`computer`" was intended.

[1] Good et al. also incorporated a kind of "dynamic" flexibility, spelling correction, into their mail system interface.

[2] However, not all of these simple lexical transformations necessarily arose from users' input errors. Damerau's collection of rejected index keywords included those caused by users' keyboarding mistakes, equipment failures (especially paper tape equipment failures), and other sources of error peculiar to Damerau's data-processing application. Subsequent researchers, however, have supported the general conclusion that a large percentage of all user input errors can be characterized as simple typographical errors, similar to those that Damerau found [6, 11, 28].

Substituting one character. In this case, exactly one of the characters in the input term was wrong. The user of the system may have entered "`compiter`" when he or she had intended to type "`computer`", for example.

Transposing two adjacent characters. The erroneous term could be produced by exchanging the positions of two adjacent characters in a known term — for instance, the user may have typed "`compuetr`" instead of "`computer`".

Because these kinds of errors are so simple, it is generally straightforward for a computer program to include appropriate error correction procedures as part of its interface. Most command language interfaces contain some kind of dictionary that defines the entire set of commands that may be entered. When the computer user enters an invalid command name, then, it is a simple matter for the interface to consult the command dictionary and locate the set of command names that are lexically similar to the unrecognized input word. (This search can run very quickly.) This is the set of the possible corrections for the user's misentered command.

Furthermore, because the above-listed types of typographical errors are so common (as a percentage of all user input errors), appropriate error correction procedures can be very useful. As Damerau described, over 80% of the index terms rejected by his data retrieval system simply had one letter missing, extra, incorrect, or transposed with an adjacent letter. Morgan [23] later incorporated Damerau's spelling correction techniques into a batch programming system. Because Morgan's improved operating system and compiler could automatically correct many common, simple errors in batch jobs, programmers made fewer fruitless attempts to submit each job — obviously an important benefit to the computer users who must wait several hours to receive the results of any batch program submission. Morgan estimated that his batch spelling correction facilities saved an average of one and a half submissions per job, thereby reducing the average number of submissions per job to two for CUPL programs and to five for FORTRAN programs.

More recently, Durham et al. [6] demonstrated that simple spelling correction techniques, based on Damerau's observations, could also be effective as part of interactive command line interfaces. Durham and his colleagues added such error correction facilities to an electronic mail system, RdMail, which was in heavy daily use by a community of several hundred people. Durham's research is therefore quite different from that of Good et al., which was previously described. Whereas Good et al. used a simulated electronic mail system in order to arrive at a "user-designed" interface, Durham and his colleagues used an actual, heavily utilized mail system in order to discover how error correction could be incorporated into an existing interface.

Durham's modified mail system ran for 41 days and recorded 23,361 sessions. In that time, RdMail discovered and processed 2,527 erroneous keywords: 2,031 unrecognized terms and 496 ambiguous abbreviations. Table 2.2 summarizes the performance of the spelling corrector on these terms.

In total, 27% of the erroneous terms were properly corrected by Durham's improved interface. Although the interface offered corrections for 44% of all the erroneous keywords (spelling corrections in 24% of the cases and disambiguations in another 20% of the cases), the RdMail users accepted these corrections only some of the time. Users accepted a spelling correction in only 66% of the cases in which one or more corrections were offered; users approved only 56% of the interface's disambiguations. The final result was that for 27% of all the erroneous input terms, the user corrected the error by choosing one of the terms offered by the improved RdMail interface. In about half of these cases, the interface offered exactly one correction which was accepted by the user.

Although this 27% overall success rate may seem low, a closer examination of the data reveals that Durham's spelling correction techniques were very good at solving the problems they were designed to address. Of the erroneous terms that were not corrected by the new RdMail interface:

Table 2.2. Results of a Command Spelling Corrector, Adapted from Durham et al. [6]

Corrected Terms		**Uncorrected Terms**	
Ambiguity	11.0%	Alphanumeric, 1–2 characters	20.3%
Typographical error:		Alphanumeric, 3+ characters:	
Missing letter	4.8%	Syntax or vocabulary error	19.8%
Extra letter	4.0%	Typographical error	2.9%
Wrong letter	4.5%	Intentional error	3.2%
Transposition	2.7%	Rejected good correction	0.4%
		Other nonalphanumeric errors	26.4%
Total	27.0%	**Total**	73.0%

- 20.3% of these terms contained only one or two alphanumeric characters. In general, it is difficult to correct short terms because they contain so little lexical information. This problem is especially severe for interfaces that attempt to correct mistyped abbreviations, as the improved RdMail interface did. For these reasons, Durham et al. disabled many of the RdMail spelling correction techniques for one- and two-character terms.

- 19.8% of the erroneous terms were due to syntax and vocabulary mistakes. In these cases the RdMail users apparently forgot the correct term (perhaps using an unrecognized synonym), omitted a required keyword from a command, or made some other sort of language error. Durham and his colleagues, however, had no expectation that their improved RdMail interface would correct these kinds of mistakes. Language-level errors cannot be addressed through spelling correction alone; one must incorporate flexible parsing into the interface, as Good et al. did [9], in order to cope with these kinds of errors.

- 26.4% of the terms were nonalphanumeric terms, which Durham's spelling corrector did not attempt to process. These terms included control characters (12.9% of all erroneous terms), numbers (4.6%), and punctuation (8.9%). Many of the control characters were apparently due to incorrect use of the `Control-s` key, which was used to suspend the output of text to users' terminals. Many of the numbers and punctuation marks appeared as the first word of an input line. However, in the RdMail command language, the first word must be the name of an RdMail command. Clearly, this is another language issue that is beyond the scope of Durham's spelling correction research.

- 3.2% of the erroneous terms were apparently entered intentionally by people who were exploring the corrector's capabilities. These items were classified as intentional errors by the researchers through manual examination of the session transcripts.

- 2.9% of the terms were caused by typographical mistakes that could not be handled by the corrector. Often, this meant that the user had omitted a space between two adjacent words. Other errors were apparently caused by system issues: the placement of the `Control` key, the ability to "type ahead," and so on. Only 8 cases were attributed to multiple typographical errors (e.g., two or more missing letters) in a single term.

- 0.4% of the terms (10 cases) were apparently properly corrected or disambiguated, only to have the user reject the interface's help.

In summary, although Durham's claimed overall correction rate of 27% may appear to be low, in fact, the RdMail spelling corrector did outstandingly well on the problems that it was designed to address. The improved command interface properly corrected 89.1% of all the typographical errors and ambiguous terms that it encountered (excepting one- and two-character terms), although some of these corrections were rejected by the users of the system. The overall success rate of 27%, then, simply indicates that language issues — the use of synonyms, flexible descriptions, and "user-centered" design techniques, as previously described — cannot be overlooked. Spelling correction is useful but it cannot take the place of careful command language design.

Durham et al. demonstrated that spelling correction and abbreviation processing can be effective components of a user-friendly command line interface. Durham and his colleagues also showed that these techniques can be incorporated into existing interfaces with relative ease. Correction techniques based on Damerau's observations can be implemented in a straightforward manner, and the results of Durham's RdMail experiment [6, page 770] confirm Damerau's observation that over 80% of all typographical errors are the result of exactly one trivial error: one letter missing, extra, incorrect, or transposed with an adjacent letter. The conclusion is that even the most obvious and simple correction techniques can handle the great percentage of all users' typographical errors and dynamically chosen abbreviations.

Other researchers have supported this conclusion. McMillan and Moran [21], for example, showed that a different but similarly simple technique could accurately recognize user-chosen abbreviations for a small set of command names. McMillan's technique was this: In order to determine which one of the command names "best" matches a user's (abbreviated) input term, a command line interface executes the following steps:

1. From the set of all command names, find the set of names that have the greatest number of characters in common with the user's input term. The order of the characters within the words are not considered; only the number of shared characters is important.

2. Remove all but the shortest command names (i.e., those names containing the fewest characters) from the set produced in the previous step.

3. From this remaining set of command names, choose the name that appears first in an alphabetical ordering of the set.

The command name selected in the final step of the algorithm is the word that "best" matches the user's original input, and therefore, this word is assumed to be the intended expansion of the user's abbreviation.

McMillan and Moran's matching algorithm is extremely simple — it does not even consider the order of the letters in the abbreviation or the command names — but even this straightforward algorithm can be surprisingly effective. In an experiment, McMillan and Moran gave a list of 17 commands to 21 college students, none of whom was an experienced computer user. Each student was asked to choose an abbreviation for each of the commands, and the researchers used the above-described technique to match these abbreviations with the original commands. The results were excellent, particularly in light of the algorithm's simplicity. For 9 of the 17 commands, 100% of the students' abbreviations were correctly recognized. The algorithm worked well for individual students as well: Of the 21 lists examined, the algorithm correctly recognized 100% of the abbreviations on 7 of the lists and recognized 94% on 7 more. Overall, 88% of the students' abbreviations were correctly paired with the full command names.

In summary, McMillan and Moran's experiment showed that even an unsophisticated algorithm such as theirs could yield surprisingly accurate results. The researchers demonstrated that their heuristic could correctly identify a great percentage of user-chosen abbreviations, and it is clear that the same heuristic could be used as a more general input matching function to both expand abbreviations and correct simple typographical errors such as those previously described. Furthermore, it could do all this with an absolute minimum of information: just the set of acceptable terms and the user's actual input.

2.1.4 Use of Context

If such a simple matching heuristic could so accurately discern users' intentions, McMillan and Moran reasoned, then it should be possible for a more intelligent technique to recognize users' intentions in practically all cases. By making use of more interaction context, a sophisticated input parser should be able to approach 100% recognition of all user inputs. McMillan and Moran suggested for instance that the accuracy of their matching function would improve if it were changed to take various lexical contexts into account. The researchers' original algorithm essentially treated words as unordered sets of characters. An improved matching algorithm, however, would consider the order of letters within words and give preference to cases in which the letters of a user's input matched in the proper order with the letters of a word from the computer's lexicon. In addition, a sophisticated matching function might notice common sequences of letters or give greater weight to character matches near the beginnings of words. Both of these improvements make use of *lexical context* — information about the environments in which individual characters are placed — that can be recognized and used to improve an interface's ability to understand the meanings of computer users' inputs.

Lexical context is information that is present within an individual word. This includes the characters that make up the word and the arrangement (order) of those characters. These things provide lexical clues that can be used to expand abbreviations and correct spelling errors. Additionally, the form of a word may sometimes provide *semantic* clues (i.e., information about the intended meaning of the word) as well. In English, for example, the suffix *-tion* almost always indicates that the containing word is a noun. Similarly, in the UNIX C shell, the prefix "`-`" typically appears only in words that are command options and the character "`/`" generally appears only within file names. Information such as this can be used to guide a shell command parser. Lexical context is only one kind of context, however. An intelligent interface can make effective use of several other levels of context as well: syntactic context, short-term interaction context, long-term interaction context, and other types of general knowledge.

Syntactic context is information that arises from the syntax (structure) of a language. In a typical command language, for example, the first word of every input line is expected to be the name of a command. This word acts as a verb, naming the action to be performed. Subsequent input words, called *command arguments*, may be expected to name files, specify options, describe input redirection, or name other objects specific to the command. Certain command arguments may be required to have special features; for example, it might be necessary that a name refer to an *existing* file. Often, different commands interpret arguments in different ways. This means that each command name (the first word on the user's input line) must be associated with a specific syntax for command arguments.

Syntactic information is obviously important for accurate error-tolerant parsing. When a spelling error is found in a command line, an intelligent parser can use syntactic information to constrain the set of possible corrections to only the syntactically acceptable alternatives. (Once the set of possible corrections is constrained, lexical information can be used to choose the best correction from the set.) It would be entirely unhelpful for a parser to correct an erroneous input term to a word that, although lexically similar, is syntactically invalid — to correct the input word to a command name when a file name is actually required, for instance.

Another type of context, the *short-term interaction context*, is defined by an individual computer user's recent inputs and the recent states of the computer system itself. The information at this level of context is different for different people and changes fairly quickly over time for any individual person.

Interaction context arises from the tasks that a computer user performs — document preparation, debugging, and so on — and how that person coordinates individual commands to complete those tasks. In general, a computer user's commands are not procedurally isolated from one another. Instead, because it is often necessary for a person to enter a sequence of several commands in order to complete a task, adjacent commands are generally related to each other. Hanson, Kraut, and Farber [12], in their study of UNIX shell commands, discovered that many shell commands fall into separate functional groups. Commands within a functional group are likely to be used in conjunction with other commands in that group. Overall, UNIX shell commands cluster into separate task-oriented groups, and these groups are tied together by orienting and process management commands. An intelligent interface can use this kind of information in order to discern or even anticipate a user's intentions; for example, when an input error occurs, the parser can be biased towards corrections in the user's current command group.

Interaction context is also useful because computer users, both novice and experienced, tend to "repeat themselves" and enter commands that they have recently entered — often with exactly the same command arguments. Like Hanson et al., Greenberg and Witten [10] studied the use of UNIX shell commands by a variety of computer users. Greenberg and Witten discovered that C shell users repeat themselves with amazing frequency. On average, for every command that a person enters, there is a 50% probability that the command is *identical* to one of the 10 commands that immediately precede it in the user's input history. (There is a 26% chance that the command has never been entered before by this person, and there is a 24% chance that the command has been entered before but is not a repetition of one of the previous 10 commands.) Even though this statistic alone is very impressive, the frequency of matches among the recent command history can be greatly improved by simply pruning repetitions from the user's command history list, maintaining separate history lists for each directory (in order to approximate task-specific command histories), and allowing for partial matches in which the user's current command is an extension of a recent previous command. Employing these additional strategies, Greenberg and Witten increased the matching frequency by 13%: In other words, there is a 63% chance that a user's command will match (i.e., be identical to or an extension of) one of the 10 previously entered commands in the user's pruned, directory-sensitive history. Even when commands are not repeated verbatim, it is often the case that new commands will refer to objects — files, for example — that were recently referenced by preceding commands. Clearly, an intelligent interface should take interaction context, both short-term and long-term, into account when discerning the intent of a user's command.

Long-term interaction context is produced by a computer user's actions over a relatively long period of time, generally spanning several sessions with the computer system. Like short-term history, long-term history can be very useful to an intelligent interface. Many researchers have shown that computer users tend to use a relatively small set of commands over and over; for instance, in their study of UNIX command usage, Hanson et al. wrote [12, page 45]: "Users of the UNIX operating system have a large number of commands available to them, and yet they used only a small proportion of these commands with any frequency. For example, although users had well over 400 commands available to them, in the process data, 10 percent of the commands accounted for almost 90 percent of the command usage." In a separate study, Greenberg and Witten [10] discovered that UNIX computer users repeat their shell commands in the long term (although repetitions most commonly occur in the short term, as previously described). Greenberg and Witten found that on average, for any shell command that a person enters, there is a 74% chance that the person has entered that command at some time in the past. That is, almost three out of every four commands are repetitions of previous actions.

Just as computer users tend to repeat their commands, they should also be likely to repeat their mistakes. An intelligent system can automatically keep track of individual users' habits — frequent actions, frequent misspellings and abbreviations, and so on — over a period of many separate computer sessions. This user profile can be saved and used to guide the interface's command interpreter in future sessions. McMillan and Moran [21] suggested that long-term interaction context for individual users could augment spelling correction and abbreviation processing by biasing the parser toward the most commonly used command names. It should also be possible for an intelligent parser to handle users' common mistakes, as Hayes et al. wrote [14, page 22]: "A graceful interface should recognize and adjust to the idiosyncrasies and preferences of its user. This includes the ability to spot and correct recurring typographical, spelling, or syntactic errors." In addition, long-term context can allow users to manually extend a command language by defining new aliases, synonyms, or abbreviations for future use.

Finally, an intelligent computer system can make use of context that arises from general knowledge in the domain of discourse. All sorts of data about the computer user, the computer system, and the human-computer interface itself can be part of this context. This information can be complex; for example, an intelligent interface might understand how separate commands can be combined to achieve certain goals, thereby allowing the computer system to analyze its user's goals and suggest more efficient ways of meeting them. General contextual knowledge can also be as simple as Damerau's rules of thumb, or as basic as knowledge of the physical arrangement of keys on the computer system's keyboard.

Spelling correction algorithms can make particular use of this last item because it is well known that the layout of a computer's keyboard greatly affects the likelihoods that certain typographical errors will occur. In one analysis of keyboarding errors, Grudin [11] concluded that most substitution errors (i.e., errors in which one letter is replaced by a different letter) involve adjacent keys. By examining a large corpus of errors Grudin showed that in 58% of all substitution errors, a correct character was replaced by a character immediately adjacent to the correct character on the keyboard. In fact, 43% of all substitution errors involved adjacent keys in the same row; only 15% involved adjacent keys in the same column. An additional 10% of substitution errors involved "mirror image" keys that have the same positions on opposite hands. Obviously, all of these statistics can be used by an intelligent, error-tolerant interface in order to choose the most likely correction for a user's erroneous input.

Each of the above-described levels of context adds to the ability of an intelligent command line interface to understand its user's intentions. An interface that corrects typographical errors, for example, might draw on all of these levels. Using the first level, lexical context, an intelligent system can select all of the vocabulary words that most resemble an erroneous input term. The second level of context, syntactic context, allows the system's command parser to use the form of the user's command to prune the set of candidate corrections to include only the most appropriate terms — command names, file names, command options, or whatever else is expected. By using short-term interaction context, the interface could limit the search to the most recently used words, or at least bias the spelling corrector to favor these recently entered terms. By consulting the user's long-term interaction history (the fourth level of context described above) the intelligent interface might examine a list of the user's common errors and abbreviations, and by using the final level of context the intelligent interface could employ general rules of thumb and inference. For example, as described previously, an interface could make use of the fact that certain kinds of errors are more common than other kinds of errors.

With so much information available, a well-designed, intelligent command language interface should be able to understand almost everything that its human operators enter [15, 21]. Moreover, existing systems demonstrate that this kind of intelligent behavior is in fact possible. One of the earliest and most widely known examples is the "Do What I Mean" or "DWIM" facility of the Interlisp programming system. The DWIM facility was an integral part of the Interlisp system and embodied the philosophy that computer systems should make intelligent interpretations of user's inputs [33]. DWIM used the levels of context described above to correct errors in Lisp programs, and at this task the DWIM facility was amazing successful. Teitelman, one of the implementors of the Interlisp system, wrote [32, page 17.3]: "We have put a great deal of effort into making DWIM 'smart,' and experience with perhaps fifty different users indicates that we have been very successful; DWIM seldom fails to correct an error the user feels it should have, and almost never mistakenly corrects an error." In short, by using DWIM, Interlisp provided an intelligent interface that effectively met the needs of its users. It is surprising that today, over twenty years after Teitelman, Morgan, and others demonstrated the effectiveness of intelligent, error-tolerant user interfaces, these kinds of command line interfaces are so rare. Fortunately, the situation is starting to improve. Computer scientists have rediscovered the importance of user interface design and have started to reincorporate intelligent features into human-computer interfaces, including that of the UNIX command shell.

2.2 The `metric` Library

The `metric` library [13], written by Hawley, was one of the earliest attempts to incorporate spelling correction into the UNIX command shell's interface. Like many other researchers, Hawley believed that in order to be most useful, interactive computer systems had to be tolerant of their users' simple input errors. Hawley wrote [13, pages 1–2]:

> One of the simplest and most obvious ways to get a computer system to "do what I mean, not what I say" is to make the system smart enough to deal intelligently with misspelled, mistyped, or abbreviated input. In an interactive command-driven environment (like UNIX)... some simple spelling correction algorithms running at the system level could have many benefits.... [W]henever such trivial spelling errors are made, the system should be able to proffer a good guess at the intended input, just as a friend looking over one's shoulder might, saving the user from the tedium of fixing trivial errors.

As a first step toward this goal, Hawley created the `metric` library. This library was a small set of C language functions that other C programs could invoke in order to correct simple errors such as those described above. At its core the `metric` library provided a function named `bestmatch`. The `bestmatch` function accepted three arguments: (1) an input string, (2) a table of strings to be matched against, and (3) a "metric" function that measured the similarity of any two strings. The `bestmatch` function then located and returned the string from the table that "best matched" the input string according to the given string similarity function. The `metric` library itself defined two different spelling metric functions that programs could give to `bestmatch`. Alternately, a program could provide its own specialized metric function.

By using the `bestmatch` function, C programs could correct users' input terms by comparing those terms against tables of acceptable inputs. Unfortunately, even though this sort of spelling correction is highly desirable, it would have taken a Herculean effort to make `bestmatch` an integral part of the UNIX system interface. In order to make direct use of the `bestmatch` function, hundreds of existing programs would have required modification, and the required changes would not have always been straightforward.

There were (and still are), however, some specialized kinds of spelling correction that could easily and usefully be incorporated into almost all UNIX programs — in particular, the correction of file names. Many programs interpret some of their command line arguments as file names. File names are generally longer than most other command line arguments and are therefore more likely to contain input errors. In addition, the general length of file names provides lexical context that makes it possible to correct errors in these arguments with accuracy. Therefore, Hawley reasoned that a function specially designed to correct file names would be widely useful. Such a function would also be straightforward to implement and incorporate into existing UNIX programs. Because most programs use file names in simple, stylized ways, it would take a minimum of effort to change existing programs to use a function that corrected file names.

For the above reasons, the `bestmatch` function was not the principal interface to the `metric` library. Instead, the most important component of the library was a function named `pfopen`. The `pfopen` function was a "polite" replacement for the function `fopen`, which is defined by the standard C language `stdio` library and which is used by almost all C programs that manipulate files. Essentially, the standard `fopen` function accepts a file name and opens that named file for reading or writing. However, `fopen` does not attempt to analyze the file name that it receives, so if that file name is mistyped or misspelled `fopen` will fail to open the desired file. Hawley's new `pfopen` function was exactly like the normal `fopen` function, except that `pfopen` examined the given file name for simple typographical errors. In other words, given a possibly mistyped file name, `pfopen` looked for an existing file name that "best matched" the possibly mistyped name. When `pfopen` detected one or more errors in its original file name, it corrected the name (possibly with confirmation from the user of the program) and then opened the corresponding file. Because `pfopen` was exactly like the standard `fopen` function except for this new ability, it was straightforward for Hawley to modify existing programs to use the `metric` library's `pfopen` function. Hawley made the necessary changes to several standard UNIX utility programs — including `cat`, `pr`, and `more` — thereby incorporating a limited amount of error tolerance into these programs with an absolute minimum of effort.

This is how Hawley improved the UNIX shell interface. As previously discussed in Section 1.3.1, most of a UNIX shell's language is defined and interpreted by programs other than the shell itself. Therefore, in order to add input error tolerance to the shell's interface, one can add intelligence either to the shell program itself or to the other programs that the shell invokes. Hawley adopted the second approach. By making existing UNIX programs more able to deal with input errors, Hawley effectively added this same ability to the shell's command line interface.

The principal advantages of Hawley's approach were twofold. First, because the `metric` library's spelling correction functions could be invoked by each program for itself, it was largely unnecessary to change the UNIX command shell. In other words, Hawley did not have to teach the shell about the syntax of every other program's command line arguments in order to implement error-tolerant command line parsing. The shell did not need to know which arguments were to be interpreted as file names and which were to be interpreted as command options or other kinds of arguments; this knowledge was contained in the individual programs themselves. This allowed the shell itself to be simple, and in addition, this approach made it easier to add new programs to the system and to modify old ones because the shell itself did not need to be updated in response to such changes. The second advantage of Hawley's approach was that it was straightforward to incorporate a limited but important kind of error-tolerant parsing into existing UNIX programs. Programmers, by simply replacing calls to `fopen` with calls to `pfopen`, could quickly enable their programs to detect and correct simple errors in file names. Other kinds of error tolerance could be implemented gradually. New programs could be written to make full use of the `metric` library functions and old programs could be rewritten over time to make use of the `metric` library features.

However, Hawley's approach toward improving the UNIX shell interface also had important disadvantages. Most significantly, there are many aspects of the shell's command language that cannot be changed without modifying the shell program itself. For example, although programs that used the `metric` library could detect and correct errors in their own command line arguments, only the shell can detect and correct errors in command names themselves (i.e., the names of the programs that the shell invokes). Hawley, of course, could have modified the UNIX shell to use the `metric` library in order to correct command names, while leaving the correction of command arguments to the other programs on the UNIX system. This illuminates a second problem with Hawley's approach; namely, that in order to fully improve the shell's command line interface Hawley needed not only to change the shell, but also to change every other program on the UNIX computer system. As just described above, it is advantageous for each program to do its own spelling correction because it is easiest to keep the knowledge of a program's command line syntax within the program itself. However, this approach is also disadvantageous because it requires one to change hundreds of existing programs — a massive undertaking. It may, in fact, not be possible to modify some existing programs due to a lack of source code or an inscrutable design.

Hawley's distributed approach to error correction with the `metric` library had one final, important shortcoming. Because each program performed its own error detection and correction, in isolation from all other programs, certain kinds of very useful contextual information were lost. For example, the `pfopen` function relied on lexical information alone in order to correct file names. Hawley described how this lack of context hindered `pfopen` [13, page 8]:

> The algorithm simply moves through the segments of the path and finds the best match for each piece. It cannot correct cases which involve a missing slash in the path (e.g., "`/usrbill`" instead of "`/usr/bill`") or cases which involve missing or added segments (e.g., "`/usr/cmd/cat.c`" instead of "`/usr/src/cmd/cat.c`").

The `metric` library's `pfopen` function did not attempt to distinguish one type of file from another, nor did it remember which files had been recently referenced (possibly by programs other than the current one), nor did it try to adapt to individual users' habits. These kinds of data could have helped address the problems described above, but because there was no persistent, central knowledge base in the `metric` library, there was nowhere to store these kinds of contextual data for later use. Hawley realized that this was a severe handicap and wrote [13, page 9] that "there is good reason to believe that more contextual information (like profiles of individual user's habits) can greatly improve accuracy and response time" in the error correction heuristics. Hawley further suggested [13, page 5] that the appropriate repository for this contextual data is within a "smart shell" that keeps track of its users' interaction histories.

VALET, the intelligent UNIX shell interface described in this thesis, attempts to implement Hawley's above suggestion. VALET incorporates error correction techniques that are similar to those in the `metric` library. Unlike Hawley's library, however, VALET combines these techniques with a large contextual knowledge base that describes such things as the sets of commands and files recently referenced by the user. This allows VALET to make corrections that are more accurate and more helpful than those that could be made by isolated programs using the `metric` library.

2.3 The `tcsh` and `zsh` Shells

Hawley's `metric` library allowed UNIX programs to perform their own spelling correction, and in this way Hawley indirectly improved the UNIX shell's command line interface.

However, Hawley realized that in order for the shell's interface to be as user-friendly as possible, spelling correction techniques and contextual information had to be incorporated into the shell program itself, not just into the other programs that the shell invokes. Following this advice, a few modern command shells for UNIX now provide simple, context-sensitive error correction facilities. Two such command shells are the popular `tcsh` and `zsh` programs.

The `tcsh` command shell [27], written by Placeway, Zoulas, and others, is a modern and enhanced version of the standard UNIX program `csh`, also known as the C shell [17]. The C shell contains a relatively complicated command parser because the `csh` input language provides a wide variety of features: command aliases, variable substitutions, file name expansions, history references, and similar abbreviation mechanisms. Unfortunately, for all of the sophistication of its parser, the C shell has very little understanding of the commands that it executes. When the C shell receives a user's input command, that command is transformed — in shell terms, the command is "expanded." Aliases and history references are resolved, values are substituted for variable names, and so on. Each of these transformations, however, is entirely syntactic; none of these facilities depends on the intended meaning of the user's input. After the user's input is transformed the C shell attempts to execute the resultant command string verbatim, again without reference to the user's intentions or the context in which the command will be executed. Because the standard C shell does not make use of relevant contextual information it is unable to help its user avoid or correct common input errors. The enhanced `tcsh` command shell, however, is somewhat more helpful to its users. It includes command completion and spelling correction facilities that enable users to avoid mistakes and correct those mistakes that do occur.

The `zsh` command shell, also known as the Z shell [7], was written by Falstad, Wischnowsky, and others, and is similar to `tcsh`. Whereas `tcsh` is an improved version of `csh`, `zsh` is a modern and enhanced version of the standard UNIX program `sh`, otherwise known as the Bourne shell. Like the C shell, the Bourne shell is a widely used, interactive UNIX command shell. Further like the C shell, the Bourne shell has sophisticated syntactic abbreviation facilities but fails to make use of contextual information that would help its users avoid and correct input errors. The Z shell adds such context-based error correction facilities to the Bourne shell in the same way that `tcsh` adds such facilities to the C shell.

2.3.1 Programmable Command Completion

Both tcsh and zsh provide two separate but related features for context-based input error avoidance and correction: *command completion* and *spelling correction*. The first of these features reduces typing errors and tedium by enabling the shell to complete partially entered words within a command. The shell user can, by simply typing the prefix of a word and then hitting a special key (in both tcsh and zsh, the Tab key), ask the shell to insert the remaining portion of the word into the command line. For example, suppose that the computer user wants to see a listing of the files in the directory named sources. The appropriate UNIX shell command for this action is "ls sources". Using the shell's command completion facility, the user can type just:

```
ls sou
```

At this point, before hitting the Return key, the user can have the shell complete the partially entered directory name. The user can press the Tab key and the shell will append the letters "rces" to the command line:

```
ls sources
```

The user can then execute this completed shell command by pressing the Return key as usual.[3] Command completion becomes more valuable as the words to be completed become longer. Because this feature reduces the amount of typing that users must do, it reduces the number of typing errors that users make.

For many years the standard C shell has been able to complete partially entered file names (as just illustrated), command names, and user login names. For each kind of completion the user simply presses the Escape key to have the shell finish an incomplete word. However, because there is no way to tell csh about the command line syntax required by the various UNIX programs, csh uses simple heuristics to decide when each kind of completion is appropriate. For example, command name completion is always used for the first word on the command line. User name completion is invoked for words that begin with the character "~" and file name completion is used in all other cases. In

[3] In the example above, it was assumed that the user's input "sou" was a prefix of exactly one file name (namely sources) in the current directory. If the prefix "sou" had not been sufficient to uniquely identify exactly one file name, the shell would have behaved slightly differently. It would have completed as much of the user's input possible or displayed a list of possible completions. Details are available in the documents that describe tcsh [27] and zsh [7].

the ordinary C shell there is no way to change these rules or specify special lexicons to be used with certain commands.

The `tcsh` shell improves on the standard C shell by providing a *programmable* completion mechanism. Using programmable completion, shell users can teach `tcsh` the command line syntaxes of various commands and specify the lexicons to be examined during completion. One particularly useful application of this feature is that one can tell `tcsh` that the `cd` command expects its (single) argument to be the name of an existing directory — not just *any* kind of file. The built-in `cd` command changes the "current directory" of the shell itself and is one of the most frequently used shell commands. The special command `complete` is used to describe how completion should be applied to other shell commands; for the example just described, a `tcsh` user would enter:

```
complete cd p/1/d/
```

This command tells `tcsh` that the first argument (the first positional argument, indicated by "`p/1`") for the `cd` command must be the name of a directory (indicated by "`d`"). A more complex example is this:

```
complete finger 'c/*@/$hostnames/' 'n/*/u/@'
```

The expression "`n/*/u/@`" specifies that arguments to the `finger` command will be completed from the set of user login names, and that when a user name is completed, the character "`@`" will be automatically appended to the name. In addition, the expression "`c/*@/$hostnames/`" indicates that immediately after the "`@`" character, completions should be drawn from the list of words in the shell variable `hostnames`. This variable must be set by the shell user and presumably contains the names of frequently consulted machines.

Clearly, programmable completion is a powerful facility. Because it allows the shell to accurately complete users' partial inputs, it reduces input errors and makes the shell easier to use. Both `tcsh` and `zsh` offer programmable completion facilities. All of the examples shown above are for `tcsh`, but they could be rewritten for `zsh` with minimal effort. Finally, note that in both `tcsh` and `zsh`, completion is controlled solely by lexical and syntactic information. Neither shell consults other kinds of interaction context (as described in Section 2.1.4) in order to complete partial inputs.

2.3.2 Spelling Correction

In addition to programmable command completion, tcsh and zsh both provide a second feature to alleviate the effects of users' input errors: spelling correction. As previously described in Section 2.1.3, an overwhelming percentage (over 80%) of all user input errors are isolated, trivial keyboarding mistakes: the insertion, deletion, or substitution of a single character in a word, or the transposition of two adjacent characters within a word. Given such a simply misspelled word and a dictionary of valid input terms, in almost all cases a command line interface should be able to correct the user's error accurately and automatically.

Both tcsh and zsh can apply this type of spelling correction to users' commands. Each shell can be told to correct only command names or to correct both command names and arguments. (The default behavior of each shell, however, is not to attempt any corrections at all. Spelling correction is a feature that must be enabled by individual users.) Correction of command names is possible and effective because each shell can accurately determine for itself the set of valid commands. Correction of command arguments is much more difficult, however, because neither the tcsh nor zsh spelling correction facilities normally have any knowledge of the command line syntax used by any commands — not even commands that are built into the shells themselves! Due to this lack of knowledge, each shell is forced to rely on lexical information alone in order to determine the domain of each command line argument (e.g., the set of all file names or the set of all user login names). Although lexical information is certainly significant and useful, it is often not sufficient; for example, it cannot be used to determine that a particular argument must name a directory, as opposed to just any kind of file.

Because lexical information is often insufficient for effective spelling correction, tcsh and zsh need additional information about the commands that they provide. They need syntactic information. In particular, *they need the kind of information that users already provide to the programmable completion facilities of each shell.* It would not be unreasonable to expect that when a user describes the command line syntax of a certain command to the shell, for use by the command completion facility, that information would also be available to and used by the spelling corrector. Surprisingly, however, this is not the case in either tcsh or zsh.

In the current versions of tcsh and zsh, the spelling correction facility does *not* consult the syntactic command line information that is provided to the command completion facility. Sadly, this flaw reduces the utility of each shell's spelling correction — in fact, this defect makes full command line checking too cumbersome to really be useful. Users of `tcsh` and `zsh` who enable spelling correction at all generally enable it only for command names, not for command arguments. Because neither shell utilizes syntactic information about command line arguments, *even when such information is available*, the spelling corrector of each shell is prone to making inappropriate and annoying attempts at correction.

Figure 2.1 illustrates how troublesome command argument spelling correction can be in `tcsh`. In the space of five commands, `tcsh` makes three nonsensical attempts to correct its user's input. The first incorrect attempt can be attributed to `tcsh`'s lack of syntactic knowledge about the arguments to the `complete` command, but the two subsequent attempts at correction violate syntactic information that was explicitly provided to `tcsh` by the user of the shell.

The first erroneous attempt occurs when the user enters a `complete` command in order to tell `tcsh` that the first argument to `cd` must be a directory name. (The `complete` command was previously described in Section 2.3.1.) Although the `complete` command is built into the shell itself, the `tcsh` command correction and completion facilities have no

`1 jaguar> set correct = all`	The user enables spelling correction for both command names and arguments.
`2 jaguar> complete cd 'p/1/d/'`	The user tells `tcsh` that the argument to `cd` must be a directory name.
`CORRECT>complete cd ./././ (y|n|e|a)? n`	`tcsh` offers to "correct" the second argument to the `complete` command to a nonsensical value.
`3 jaguar> complete mkdir 'n/*/d/'`	The user tell `tcsh` that all of the arguments to `mkdir` must be directory names.
`4 jaguar> touch file`	The user creates a file named `file`...
`5 jaguar> mkdir files`	...and a directory named `files`.
`CORRECT>mkdir file (y|n|e|a)? n`	`tcsh` offers to "correct" the argument to an invalid value.
`6 jaguar> cd filed`	The user makes a typographical error: "`filed`".
`CORRECT>cd file (y|n|e|a)? n`	Again, `tcsh` offers a nonsensical "correction."

Figure 2.1. An Annotated Transcript of `tcsh` Spelling Correction

predefined knowledge of the command line syntax that the `complete` command requires. Without this knowledge, `tcsh` automatically (and incorrectly) assumes that all of the arguments must be file names. As shown in the transcript, `tcsh` was unable to find a file name that was lexically similar to "`cd`" so no correction is offered for that argument. However, the second argument "`p/1/d/`" is handled slightly differently. After discovering that "`p/1/d/`" is not the name of an existing file, `tcsh` decided that the file name "`./././`" (which is a *very* unusual way of referring to the shell's current directory) was lexically similar to the original "`p/1/d/`" argument, and so "`./././`" was offered as a correction to the user. Unfortunately, because the argument "`p/1/d/`" was not intended to refer to a file, the correction was entirely inappropriate and almost certainly unexpected by the user.

The second erroneous attempt to correct the user's input occurs when the user enters the command "`mkdir files`" to create a new directory. Although `tcsh` was previously told by the user that the arguments to `mkdir` should name directories,[4] the `tcsh` spelling correction facility ignored this information and suggested that the argument "`files`" should be corrected to "`file`" — the name of a regular file! Obviously this suggestion is entirely unhelpful to the user. Likely, the real effect of this attempt at correction is simply to annoy the user.

The third and final erroneous correction occurs when the user enters the command "`cd filed`" which contains a typographical mistake. The word "`filed`" is lexically very similar to both `file` and `files`, and either would be a reasonable correction based on lexical data alone. However, because the argument to `cd` must name a directory, `files` is the much superior alternative; because `file` names a regular file and not a directory, the command "`cd file`" is nonsense. Unfortunately this is the correction that `tcsh` offers. Even though the user had previously told `tcsh` that the argument to `cd` must name a directory, `tcsh`'s spelling correction facility ignores this knowledge and relies on lexical similarity alone to match the input word "`filed`" against the names of all the files in the

[4]In truth, the command line arguments given to `mkdir` should *not* name existing directories, so the information about `mkdir` provided to `tcsh` in Figure 2.1 is not entirely accurate. The purpose of `mkdir` is to create new directories and the arguments given to `mkdir` are the names of the directories to be created. Unfortunately, using the `complete` command, it is not possible to tell `tcsh` that an argument should refer to a nonexistent directory. The lie that `mkdir`'s arguments should be (existing) directory names is pragmatic, however, because it allows `tcsh` to complete the directory components of pathnames given to `mkdir`. In any case, the point is this: Even though the information provided to `tcsh` about `mkdir` is not entirely accurate, it is nonetheless inappropriate for `tcsh` to subsequently ignore this user-supplied information when it attempts to correct its user's `mkdir` input commands.

shell's current directory. That `tcsh` ultimately chooses to offer the correction "`cd file`" instead of "`cd files`" is because, given a choice, the `tcsh` spelling corrector prefers to assume that the user made an insertion error (i.e., typed one extra character) rather than a substitution error (i.e., typed one incorrect character).

As the transcript in Figure 2.1 illustrates, the spelling correction facility in `tcsh` leaves much to be desired. In fact, `tcsh`'s correction of command line arguments is so often unhelpful and cumbersome that most `tcsh` users disable the feature entirely. The `zsh` input correction facility suffers from the same problems found in its `tcsh` counterpart because both shells rely on lexical information alone to make corrections in users' inputs. Neither shell can correct command line arguments with great accuracy because neither the `tcsh` nor `zsh` spelling correction facilities uses any knowledge of the command line syntax used by any commands. Each shell is forced to infer the domain from which each command line argument is drawn (e.g., the set of all file names or the set of all user login names), and lexical information alone is often insufficient for this task.

VALET addresses the above-described shortcomings in the `tcsh` and `zsh` spelling correction facilities. Unlike the `tcsh` and `zsh` shells which are normally ignorant of the command line syntaxes used by all commands, VALET incorporates a large, predefined knowledge base that describes the syntaxes used by many of the most commonly used UNIX commands. For example, VALET automatically knows that the argument given to a `cd` command must name an existing directory, and that the arguments given to `mkdir` must name nonexistent directories. VALET can also make use of file name extensions in order to guide its input corrector. VALET knows, for example, that the names of input files for the `tex` and `latex` programs generally end with the characters "`.tex`". Not only does VALET use lexical and syntactic information to correct users' input errors, but it also refers to its users' short- and long-term interaction contexts in order to make more sophisticated kinds of corrections. VALET can, for example, often determine when a referenced file is actually in a different directory than the one that the user indicated. Finally, VALET offers a command completion facility like the one included in the standard C shell. Although VALET does not offer a *programmable* completion facility like those found in `tcsh` and `zsh`, VALET's input correction facilities are much more sophisticated than those found in today's popular UNIX command shells.

2.4 SAUCI, the Self-Adaptive User-Computer Interface

As just described, the tcsh and zsh command shells improve upon the standard UNIX shells while preserving the traditional command line interfaces of those programs. By providing command completion and spelling correction facilities, tcsh and zsh both reduce the frequency of input errors and mitigate the effects of those errors that do occur. However, even with these improvements, tcsh and zsh still do relatively little to help their users input appropriate, well-formed commands. Although tcsh and zsh can correct (within the limits previously discussed) inputs once they have been entered, neither program can effectively describe the commands and command options that are available to the user.[5] Neither can they offer any assistance to the user in formulating a plan of action; both tcsh and zsh are oriented toward individual commands and understand nothing of their users' overall tasks and goals. The authors of tcsh and zsh are hardly blameworthy for the omission of task-specific guidance in their shells, however, because neither program was intended to incorporate such novel features. The tcsh and zsh shells were designed to offer significant but only incremental improvements over previous UNIX shells. The SAUCI system, however, was designed to replace the traditional UNIX shell interface altogether.

SAUCI, the Self-Adaptive User-Computer Interface system written by Tyler and Treu [34, 35], is a research prototype that demonstrates a great variety of modern user interface design concepts. These ideas include:

- a graphical, multiwindow, form-based interface;
- adaptation to individual users;
- context-specific advice and assistance; and
- orientation toward specific high-level tasks.

[5]Once a command name is entered by the shell user, both tcsh and zsh have the ability to print a help message for that command. However, such documentation is generally either nonexistent or inappropriate for quick reference (e.g., the full UNIX "man page"). Neither shell can answer specific questions posed by the user or tailor its help messages to the user's level of experience.

2.4.1 A Graphical User Interface

SAUCI provides a graphical user interface to the UNIX shell. The interface presents several separate windows to its user, and these windows make it clear that SAUCI divides the process of command entry and execution into a sequence of unique steps or phases.

The first phase is the *prompt phase.* To select a command to be executed, the SAUCI user first chooses the command name from a menu in the "Commands" region of the display. SAUCI has knowledge of the 50 or so most commonly used UNIX commands (as determined by Hanson, Kraut, and Farber [12]), and these commands are organized into functional groups: communication commands, file and directory manipulation commands, and so on. Each functional group of commands appears within its own menu.

Once the user has chosen the command to be executed, the SAUCI system enters the *parameter phase.* A new set of windows appears on the display, allowing the user to specify the arguments for the command. The contents of these windows are, of course, tailored specifically to the options available for the just-selected command. One window allows the user to control the "simple" options — those that are simply enabled or disabled and which require no further specification. Additional windows present the more complex command arguments. For each argument requiring text entry (e.g., a file name), SAUCI creates a separate window and presents it to the user along with appropriate instructions. Together, all of these parameter phase windows constitute a form that enables the user to view and specify the arguments for the selected command.

When the user finishes entry of the command arguments, SAUCI enters its *system response phase.* SAUCI constructs a valid shell command from the user's specifications and submits that command to the UNIX shell. The command and its output appear in a terminal-like window in the "System Response" region of the display. If the just-executed command is interactive, the user can communicate with the process through this system response window. When the command is complete the user returns to SAUCI's prompt phase in order to specify and execute another command.

2.4.2 Adaptation to Individual Users

In addition to providing a graphical interface to the UNIX shell, SAUCI *adapts* this interface to individual users, according to their experience levels and previous interactions with SAUCI. The importance of adaptation, as previously described in Section 1.2, is that through customization SAUCI can intelligently satisfy the differing needs of different people. By tailoring its own presentations according to individual users' profiles and behaviors, the SAUCI system presents a more cooperative and user-friendly interface to the UNIX shell.

Both the prompt phase and parameter phase described in Section 2.4.1 are tailored to individual users of the SAUCI system. In the prompt phase, the contents of the command selection menus are determined by rules that operate on the current user's profile. For instance, SAUCI can choose to display verbose or terse text in its menus based on the number of times that the corresponding UNIX commands have been successfully invoked by the current user. Alternately, SAUCI can choose to include or omit commands from its menus based on the user's predetermined "class" — novice, computer science student, or system programmer, for example.

The parameter phase of the SAUCI dialogue is similarly customized according to the current user's experience with the selected command. SAUCI automatically decides which command options and arguments to present and how those options should be described, depending on the user's profile and interaction history.

2.4.3 Context-Specific Advice

The user's interaction history also guides SAUCI's presentations of its own integrated command documentation and error messages. SAUCI includes built-in synopses of each of the shell commands that are known to the system. For each command these summaries describe the purpose of the command, provide brief instructions for use of the command, offer examples and special warnings, and include other similar information. Not surprisingly, SAUCI tailors these help texts according to the inferred needs of the person currently using the interface. SAUCI can provide verbose descriptions to novices and brief descriptions to experts. If the current user has previously made mistakes with a certain shell command, then SAUCI's online summary of that command will include warnings tailored to the user's past errors.

These same types of customization apply to SAUCI's own error messages. After a user has completed the prompt and parameter phases of command entry, SAUCI examines the user's command for a variety of errors — for example, SAUCI may verify that the files referenced in the command actually exist and are of the required types. SAUCI can also determine whether or not the user's command is appropriate for the current high-level task. (Task orientation is described in Section 2.4.4.) If any errors are discovered in the user's command, SAUCI presents a description of the errors, tailored according to the user's experience and current situation, and then allows the user to correct those errors.

The user does not need to respecify the entire command from scratch; rather, the user only needs to edit the contents of the appropriate parameter entry windows as described in Section 2.4.1 and then resubmit the command to SAUCI.

In summary, SAUCI makes use of specific users' interaction contexts in order to offer context-specific advice that is appropriate to its users' experiences.

2.4.4 Orientation Toward High-Level Tasks

The final user interface innovation that SAUCI includes is task orientation. Traditionally, human-computer interfaces have been organized around the individual *actions* that people may carry out with the underlying systems, with little or no attention paid to how those separate actions may be combined in order to accomplish the user's overall *tasks*. The responsibility for mapping high-level tasks (e.g., the preparation of a letter) onto sequences of actions offered by a computer system has, until recently, always been placed entirely on the user of the system. Unfortunately, this responsibility can be a significant barrier to inexperienced computer users who may not be familiar with all of the features of the systems that they use — or who may be intimidated by the wide variety of actions available in a complex computer system like UNIX. With the increasing use of computers by people who are not computer specialists, user interface designers realized that in order to be most user-friendly, systems needed to help their users map specific tasks onto sequences of coordinated actions. In other words, user interfaces needed to provide task-specific guidance.

The SAUCI prototype can provide guidance for two different high-level tasks: the preparation and printing of a text document and the creation and testing of a computer program in the C programming language. SAUCI represents these tasks by inverted tree-like structures. Each task is decomposed into an ordered sequence of subtasks; each subtask may itself be decomposed into further ordered subtasks or into one or more shell commands that must be executed by the SAUCI user in order to complete the task.

Once the SAUCI user has chosen either the writing or programming task, SAUCI presents three separate windows in the "Task Guidance" region of the display:

- The first window contains brief, step-by-step English instructions for completing the task. When the user initiates a new task, this window outlines only the top-level subtasks. The user can request a more detailed set of directions at any time;

these directions are, not surprisingly, tailored according to the user's profile and interaction history.

- The second window displays a graphical representation of the selected task and the user's progress through it. The task components are organized in an inverted tree structure; the task as a whole is represented by a text label at the top of the tree. Line segments connect the task label to the labels for the top-level substeps in the task; beneath each of these steps are the constituent substeps, and so on until at the bottom of the tree are labels that represent the shell commands to be executed. The label that represents the user's current position in the task is highlighted and the labels for all the previously completed steps are boxed. This display allows the SAUCI user to track his or her progress through the task.

- The third and final window graphically depicts the files that are involved in the current task. The files are represented by text labels, each label containing the corresponding file name and a brief English description of the file contents (e.g., "misspelled words"). The relationships between files are represented by line segments that connect the appropriate labels.

People who have little experience with a certain task can clearly benefit from task-specific guidance because it enables them to work with reasonable proficiency and with minimal assistance from human tutors. In addition, people who are already experts with a task can benefit from the additional orienting context that SAUCI keeps on the display. Task-specific direction is an integral part of SAUCI's user-supportive nature, especially since such direction is tailored to the requirements of individual users. As with all of SAUCI's other displays, the information within the task guidance windows is controlled by rules that customize the presentations to the current user's individual profile and history.

2.4.5 SAUCI Results

In summary, SAUCI is a prototype research system that provides a graphical, task-oriented, context-sensitive, and user-adaptive interface to the UNIX shell. In order to assess the value of the SAUCI system, Tyler and Treu recruited three students to perform a large set of tasks with SAUCI [35]. These tasks exercised most of the basic UNIX commands with which SAUCI is familiar, and in addition there were several document preparation and programming exercises. (These are the two high-level tasks for which SAUCI can provide guidance as described in Section 2.4.4.) Tyler and Treu recruited three additional students to carry out the same set of tasks with a more traditional terminal-like interface to the shell. All six students had significant experience with VAX/VMS but little with UNIX.

The result of the researchers' experiment was that the students who used SAUCI made far fewer errors than those who used the traditional shell interface. (For this experiment, Tyler and Treu counted as errors all user actions that did not achieve the goal of the exercise at hand.) Furthermore, the subjects who used SAUCI completed the set of tasks in less time than did those who used the textual interface. Tyler and Treu concluded [35, pages 323–324]:

> [U]sers of the SAUCI interface generally did better on both measures of performance, making only about half as many errors in each stage and taking much less time to complete the tasks for all but the simple commands.
>
> ... The SAUCI users came to rely on the help system and the File System Window to achieve their goals, and mostly made errors involving confusing subdirectories with files and omitting needed argument prefixes. Users of the standard interface, on the other hand, made more kinds of errors, including misordering command arguments, misspelling arguments, and losing track of their current location in the file system.

Although the researchers noted that their experiment was too informal to be conclusive, they suggested that each of SAUCI's user interface innovations played a role in SAUCI's apparent success. The graphical, form-based interface for command specification both provided important orienting information to users and kept users from entering syntactically malformed commands (because SAUCI formatted the final textual command line itself). Task-specific guidance and context-sensitive assistance also played important roles in helping SAUCI's users complete their tasks successfully and efficiently. There was less direct evidence that adaptation to individual users contributed to SAUCI's usability, but the researchers noted [35, page 324] that the students who used SAUCI expressed "growing comfort with the interface and generally felt that it had become easier to use after the first few sessions." Tyler and Treu also suggested that SAUCI's online advice and assistance features were so well received by users because these interface components adapted to users' individual behaviors.

The ultimate goals of SAUCI and VALET, the UNIX shell interface described in Chapter 3, are the same: to provide an "intelligent" and user-supportive interface to the UNIX shell. However, of all the UNIX shell interfaces described in this chapter, SAUCI is the least similar to VALET. Even though the two systems' goals are identical, the methods by which these systems attempt to meet their goals are very different.

Most obviously, SAUCI replaces the traditional UNIX shell command line interface with a graphical, multiwindow display. VALET, on the other hand, preserves the shell's command line interface. Each approach has its advantages. By abandoning the shell's command line interface, SAUCI is able to provide important contextual information to its users; for instance, SAUCI can display the arguments for a particular command in a specially tailored form. Moreover, because SAUCI constructs the ultimate command line itself, SAUCI eliminates a large number of users' typing errors. VALET, however, requires its users to enter commands through the shell's traditional, textual interface. Under normal circumstances VALET is invisible to its users; VALET makes itself apparent only when it detects an input error. This has advantages for people who are already familiar with UNIX: These users may find SAUCI's graphical user interface to be too intrusive. Sophisticated users of the UNIX shell make frequent use of the shell's special features — file name patterns, command completion, and command history, for example — which are most easily invoked through the standard command language interface. (The unique strengths of command language interfaces were previously described in Section 1.1.) Whereas SAUCI disguises these already-existing and useful UNIX shell features, VALET provides direct access to them.

A second difference is that SAUCI incorporates its own help texts for users. SAUCI users can ask for help on specific shell commands and SAUCI automatically adapts this documentation to the experience levels and behaviors of its users. VALET, on the other hand, does not provide any help facility at all to its users. People who use VALET must rely on the UNIX documentation facilities which already exist (i.e., the `man` command). A final difference between SAUCI and VALET is that SAUCI provides some measure of task-oriented guidance. Although the SAUCI prototype system understands only two very narrow tasks, Tyler and Treu demonstrated that the idea of task-oriented assistance can be very useful, especially to novice users. VALET, however, is designed to interpret only users' commands, not users' goals. VALET is able to detect syntactically or semantically erroneous commands — and offer reasonable corrections for those commands — but it cannot detect circumstances in which a command is erroneous because it fails to fulfill the user's intention.

Both SAUCI and VALET provide "intelligent" user-adaptive interfaces to the UNIX shell. The kinds of assistance that these systems provide, however, are very different. It would be interesting to incorporate some of SAUCI's features into VALET at some future time; ideas for improving VALET are presented in Chapter 5.

2.5 SUSI, the Smart User System Interface

VALET was preceded and influenced by all of the systems described above: the `metric` spelling correction library, the `tcsh` and `zsh` command shells, and the SAUCI shell interface. Of all the systems that preceded VALET, however, the most similar is SUSI, the Smart User System Interface created by Jerrams-Smith [16].

Like SAUCI, the SUSI system is a research prototype interface to the UNIX shell program. Unlike SAUCI, however, SUSI is a "transparent" agent between the user and the shell. A person who uses SUSI communicates with the underlying shell through the usual command line interface. SUSI interrupts the dialogue only when it detects an error or a need to offer advice — in this way SUSI is very similar to VALET.

2.5.1 An Analysis of Users' Errors

Before implementing SUSI, Jerrams-Smith studied the behavior of novice UNIX users in order to understand the mistakes that such users make. Fifty-five university undergraduate students were recruited. Each was asked to learn to use UNIX well enough so as to be able to perform some simple tasks: create, format, and print a brief essay and write a small computer program. (The subjects were not constrained to these tasks, however.) All of the subjects were familiar with computer science and with other operating systems, but none had experience with UNIX. At the start of the experiment the researchers provided a brief lecture on UNIX to the students. The students then used the UNIX system largely on their own. During the experiment the researchers recorded all of the users' inputs to the shell, along with additional context such as the time at which each command was entered and the set of files that were in the shell's current directory. In addition, the students spoke into tape recorders as they worked. They were asked to describe what they were trying to do, how they were trying to do it, and what their reactions were to the shell's execution of their commands. These verbal transcripts provided valuable insights into the students' intentions and reactions.

The students learned to use UNIX over a four-week period. After these behavior study sessions were complete, Jerrams-Smith and her colleagues examined the recorded session transcripts, located the students' errors, and finally classified all of these errors according to cause. The principal purpose of this exercise, in addition to simply gathering data on users' input mistakes, was to understand how the examiners themselves recognized and inferred the causes of errors. By understanding how human experts interpret input errors, Jerrams-Smith was later able to incorporate the examiners' knowledge into an expert system within the SUSI shell interface.

The experts' classification of the test subjects' errors is summarized in Table 2.3. The researchers enumerated 135 errors in the transcripts from the first two weeks of the study and 416 errors from the transcripts of the second two weeks. The kinds of errors that the students made changed as they gained experience, as Table 2.3 shows. The causes of error included the following:

Inability to enter a command. Due to basic misunderstanding of the interface or its documentation, the user was unable to enter the command properly. For example, this category includes inputs in which the user omitted a space between the command name and the subsequent command argument. It also includes inputs in which the user typed metacharacters from the system documentation (e.g., "`<cat f1>`"). Although the novice users in the study initially made many errors of this variety, this source of error disappeared very rapidly as the users gained experience with UNIX.

Table 2.3. Summary of Errors in Novice Users' Commands, Adapted from Jerrams-Smith [16]

	Percentage of All Errors During		
Error Category	**Weeks 1–2**	**Weeks 3–4**	**Entire Study**
Inability to enter a command	29.6	3.4	9.8
Failed request for help	12.6	9.4	10.2
Use of previous knowledge	15.6	7.7	9.6
Spelling or typing error	14.8	18.5	17.6
User misunderstanding	3.7	12.0	10.0
Inefficient use of commands	4.4	15.1	12.5
Unknown input	0.0	13.7	10.3
Other causes	19.3	20.2	20.0
Total	100.0	100.0	100.0

Failed request for help. The user tried to summon online assistance but failed.

Use of previous knowledge. Errors in this category include commands that are available on other computer operating systems but not on UNIX. The subjects in the behavior study had experience with other operating systems and tried to transfer their knowledge onto UNIX. As the users became more familiar with UNIX, this source of input error became less significant.

Spelling or typing error. The input command contained a simple spelling or typographical error; for instance, one subject typed "`mial`" when "`mail`" was actually intended. As Table 2.3 illustrates, simple input mistakes were a large source of error throughout the study. In fact, spelling errors were the single most common input errors in the study. The researchers counted 97 typographical errors in the transcripts — the next most common type of error, inefficient use of commands, had only 61 instances [16, page 274]. In the second half of the study almost one-fifth of all errors were typographical slips. Additionally, in contrast to the three previous categories of error, spelling mistakes became a *more* significant cause of input errors as students gained experience with UNIX. That is, although experience quickly reduced the frequency of knowledge-based errors, the students continued to make typographical errors at a steady rate.

User misunderstanding. The command suggested that the user had an incorrect mental model of the system. For example, the user may be confusing the concept of pipes, which relay data between processes, with the concept of redirections, which relay data between processes and files. This source of error became more important as the students gained UNIX experience and perhaps attempted more complex commands.

Inefficient use of commands. This category applies to input commands that, although otherwise correct and effective, demonstrate that the user is not making the most efficient use of the UNIX system. For instance, the user may be using a sequence of `rm` commands to delete several files — "`rm file-1`" followed by "`rm file-2`" — when it would be more efficient for the user to enter a single `rm` command to delete all of the files at once.

Unknown input. This category includes commands for which the researchers could not infer any intention. Examples in this category include "`djnf`" and "`yoo`" and other nonsensical inputs.

Other causes. Jerrams-Smith and her colleagues enumerated several other minor categories for input errors [16, pages 272–273]; these categories have been combined under the heading "Other causes" in Table 2.3. Among these other classifications of errors are apparent guesses, obscenities, mistakes caused by incorrect or misleading documentation, and general lapses of attention.

From this analysis of novice users' errors, Jerrams-Smith and her colleagues concluded that a supportive interface to the UNIX shell would need to address several distinct causes of errors. Certainly, an ideal shell interface would correct trivial input errors — including typographical errors — reliably and automatically. At a higher level, an ideal shell interface should detect its users' misconceptions about UNIX and offer appropriate help as quickly as possible in order to prevent users from making the same mistakes over and over again. This means that not only should the interface recognize explicit requests for help from its user, but that the interface must in addition provide *active* assistance when required. Finally, an ideal UNIX shell interface should be able to recognize situations in which a person is utilizing the system in an inefficient way. The interface would then offer advice on how commands could be invoked in a more effective manner.

2.5.2 The Design of SUSI

Based on the preceding analysis of novice users' errors and the causes of those errors, Jerrams-Smith implemented the SUSI system to recognize and address the most common difficulties that people have in using the UNIX command shell. In order for SUSI to meet this goal it incorporates many features; most importantly, SUSI contains a knowledge base of the most common UNIX commands, a knowledge base for diagnosing the causes of users' errors, and an active assistance capability that allows SUSI to provide tutor-like advice to its users. This advice is tailored according to individual users' previous experiences and behavior patterns.

SUSI is a "transparent" agent that mediates the dialogue between the UNIX shell and its human user. That is to say that SUSI is largely invisible until it detects a need to assist the user of the shell. A person using SUSI sees the traditional command line interface to the UNIX shell, with one small difference: The files in the current directory are listed above the shell's prompt. This list is updated as necessary. By providing this listing SUSI provides important contextual information to its users. The directory listing also provides feedback; users can, for example, immediately see the results of a `cp` or `rm` command.

When the SUSI user enters a shell command, the command is sent to SUSI, not directly to the underlying shell. SUSI receives the command line, divides it into separate components (tokens), and applies "intelligent" spelling correction to each piece [16, page 283]. The next step is to parse the command. SUSI knows all of the commands that are available to the user but has detailed knowledge of only the 40 or so most frequently invoked UNIX commands [12]. SUSI's knowledge of these commands includes the required syntax for the command line arguments and information about each argument, such as whether it is required or optional, whether or not it may be repeated, and what kind of entity (if any) it must name — a file, a user, or something else. In addition, for each command SUSI keeps a list of synonyms and alternate names; these are names which novices might use in an attempt to invoke the command. SUSI does not accept these alternate names as commands, but it does use them in order to provide tutorial assistance to the user.

SUSI then consults its knowledge base in order to detect and analyze input errors. The expert system component of the SUSI system contains approximately 70 production rules that embody the knowledge gleaned from the user behavior study described in Section 2.5.1. The methods that the examiners used to diagnose users' errors are manifested by forward chaining rules in SUSI's knowledge base. Some of these rules attempt to recognize users' misconceptions about UNIX. For instance, SUSI includes a rule of the following form [16, page 280] that attempts to determine when the user has confused the concepts of output redirection (invoked by "`>`") and pipes (invoked by "`|`"):

If	"`>`" is present in the user's command
	and "`>`" is not followed by the name of an existing file
	and "`>`" is not followed by a mistyping of a file name
	and "`>`" is followed by a UNIX command
then	the user may be confusing "`>`" with "`|`".

Other rules attempt to diagnose inefficient use of commands or recognize explicit requests for help. Rules can also consult the current user's profile and history, so, for example, some rules may apply only to novices or only to experts.

SUSI invokes its rules for every input command and updates its model of the current

SUSI user accordingly. When an error or misconception is detected, the interface offers immediate assistance to the user. SUSI describes the problem and the user can request additional instruction. Although most of SUSI's explanations are fixed (and targeted for novices), portions of the texts are adapted to the user's current situation. For instance, tutorial examples will refer to files in the user's current directory.

Finally, if the command is well-formed and unlikely to cause harm (e.g., the accidental deletion of files), SUSI submits the command to the UNIX shell. Additional tutorial information from SUSI, if any, appears after the output of the command but before the next shell prompt.

2.5.3 SUSI Results

In order to evaluate the effects of SUSI, Jerrams-Smith conducted a second study of novice user behavior. Thirteen undergraduates, none familiar with UNIX, were recruited and divided into two groups; six students used the SUSI system and seven used the normal UNIX shell interface. The researchers designed a set of tasks that exercised a variety of frequently used UNIX commands and concepts (i.e., those that the SUSI system supports). The researchers then gave this task list to each student in the study and asked the subjects to carry out the specified tasks, and only those tasks, in the order specified. Jerrams-Smith and her colleagues monitored the performance of the students through the same methods used in the previous behavior study described in Section 2.5.1.

Not surprisingly, the students who used the supportive SUSI system demonstrated superior performance. Jerrams-Smith wrote [16, page 288], "When compared with the control group [using the normal UNIX shell interface], the experimental group [using SUSI] showed a significantly improved ability to use UNIX easily and efficiently." In particular, those who used SUSI excelled in three respects:

- Those who used SUSI both attempted and successfully completed more of the assigned tasks than did those who used the standard shell interface. Most of the students who used SUSI attempted all 30 of the assigned tasks, whereas those who used the normal shell attempted only 22 or 23 on average.

- The people who used SUSI worked more efficiently. That is to say that although the SUSI users completed a larger number of the assigned tasks, they entered far fewer commands than did their counterparts who used the unassisted shell interface. In fact, SUSI users entered on average only 55 commands whereas users of the standard shell entered over 110 commands on average — twice as many.

- SUSI users made fewer mistakes than did those who used the standard shell. Students using the normal shell interface frequently made the same kinds mistakes over and over. Often, several erroneous commands arose from a single underlying misconception. Students using SUSI, however, received immediate assistance when their misconceptions became apparent. Because SUSI offers tutoring in direct response to errors, the students using SUSI received critical feedback instruction that prevented future mistakes.

Jerrams-Smith concluded that the SUSI prototype system successfully met the goal of actively and intelligently supporting users of the UNIX shell [16, page 289]:

> [SUSI] helps novices to overcome the initial difficult stage of using UNIX and enables them to use it more efficiently. The interface allows experts to interact with UNIX in almost their usual way, but offers advice and guidance when it detects a novice error. Novices show much less confusion because their underlying problems are usually solved quickly and although there are some occasions when the interface is unable to help, the novice is no worse off in that situation than is the usual novice user of UNIX.

Both SUSI and VALET are research prototypes of supportive shell interfaces. In fact, VALET is remarkably similar to SUSI in design. Both systems are "transparent" agents, implemented in Lisp, that mediate communication between their users and the UNIX shell. Both systems have knowledge of all available UNIX commands and detailed knowledge of the most commonly used commands which allows these interfaces to parse and correct users' inputs. VALET and SUSI both provide spelling correction and actively assist their users when they detect the need to do so.

However, each system also has features that the other lacks. SUSI, for example, is able to diagnose novice users' misconceptions about UNIX and offer tutorial information in order to correct those misunderstandings. SUSI maintains user models that describe what its users appear to know and not know about UNIX. VALET, on the other hand, generally assumes that its users are familiar (but not necessarily experts) with UNIX and therefore concentrates on addressing the mistakes that more experienced users make — in particular, typographical errors and errors based on interaction context. VALET incorporates powerful input correction facilities but does not attempt to tutor its users in effective use of the UNIX shell. It would be interesting and useful to incorporate more of SUSI's knowledge into VALET, along with other improvements such as SUSI's omnipresent listing of the files in the shell's current directory. These and other ideas for improvement to VALET are described in Chapter 5.

VALET shares features with each of the shells and shell interfaces described in this chapter: the `metric` library, the `tcsh` and `zsh` UNIX shells, SAUCI, and SUSI. These systems preceded and influenced the design of VALET, and that design is the topic of Chapter 3.

CHAPTER 3

VALET

To a very large degree, the perceived nature of an interactive computer system is defined by the user interface of the system's command shell. A *command shell* as previously described in Section 1.3 is a special program that allows its users to invoke other programs and otherwise coordinate a computer system's resources. In general, when a person begins an interactive session with a computer, the computer system automatically starts a command shell for that user. This shell is the program with which the user initially interacts. Further, the user and shell communicate continually throughout the user's session. The shell allows its user to invoke other programs, and although those programs may temporarily take control of the user's terminal, the shell interface always reappears when those other programs have finished. Because it is the shell that is present when the computer is otherwise "doing nothing," and due to the command shell's special role in the computer system, the shell's interface has an enormous impact on users' perceptions of the computer system as a whole. The shell is seen as the fundamental interface of the computer system.

The user-friendliness of a system's command shell therefore reflects upon the system as a whole. Unfortunately for users and manufacturers of UNIX systems, the most common UNIX command shells are largely user-*unfriendly* — and therefore, UNIX systems as a whole are widely perceived to be unfriendly and hard to use, even when these systems offer a wide variety of other, more user-friendly applications. The UNIX shell interface has been widely criticized for its uncooperative nature [8, 25]. Novice users are often confused by obscure command names and frustrated by the system's lack of feedback and inability to provide intelligent advice. Experienced users are frustrated by the shell's inability to detect or correct even the most trivial errors in input commands. All UNIX users would benefit from a shell that could intelligently interpret its users' commands and then act according to its users' intentions, even when the actual input commands are incorrect in some way. In other words, people would profit from a command shell that attempted to do what its users *mean to say*.

Although the state of today's popular UNIX shells is problematic for users of UNIX systems, the situation provides an obvious opportunity for human-computer interface researchers to experiment with techniques for improving user interfaces — both the UNIX shell interface in particular and human-computer interfaces in general. The UNIX shell is ripe for experimentation: Not only does its interface badly need to be improved, but the shell is also widely and regularly used, especially within the academic computer science community. Because so many people use the UNIX shell on a daily basis, improvements to the UNIX shell are valuable, and it is easy for user interface researchers to find human test subjects in order to evaluate such improvements.

Several previous attempts to improve the UNIX shell interface were described in Chapter 2. The remainder of this thesis presents VALET, a new "intelligent" user interface for the UNIX C shell.

3.1 The Goals and Limitations of the Interface

In a sentence, the purpose of VALET is to provide a user-supportive interface to the UNIX C shell [17] that meets the needs of relatively experienced users of that shell. By meeting this specific goal the VALET interface also serves a greater purpose: namely, to demonstrate that the ideas embodied in user-supportive, cooperative, and "intelligent" interfaces are useful, effective, and worthy of incorporation into other human-computer interfaces.

VALET caters to practiced C shell users because it adds intelligent command analysis and correction to the standard C shell command line interface without fundamentally altering that interface. In other words, VALET is an *intelligent* but largely *transparent* agent.

3.1.1 Intelligent Command Processing

VALET is *intelligent* in the sense that it uses a great deal of knowledge in order to accurately interpret — and when necessary, correct — its users' input commands. VALET uses interaction context, as described in Section 2.1.4, in order to accomplish this task. This context includes knowledge of the full set of available commands (with a few minor exceptions), detailed knowledge of the most commonly used commands, knowledge of the file system, and knowledge of specific users' interaction histories, both short-term and long-term. Because VALET maintains all of this context, it can accurately detect and correct the most common mistakes that experienced shell users make: typographical errors, misspellings, errors of location (e.g., the use of a partially incorrect file name), and other minor syntactic errors.

3.1.2 Transparency

VALET is *transparent* in the sense that it makes minimal modifications to the command line interface offered by the standard UNIX C shell. People who use VALET interact with the C shell as they normally do through a textual, terminal-like interface. The shell prompts for input and the user enters a command according to the regular C shell input language. The output generated by the command, unadulterated by VALET, appears in the terminal-like window and then the process repeats with the shell prompting the user for additional commands. VALET interrupts this dialogue only when it detects an error in an input shell command. At that point VALET intercedes and takes appropriate action (i.e., describes the mistake and offers a reasonable correction if possible).

VALET's transparent, generally passive nature has both advantages and disadvantages. Part of VALET's transparency is that it preserves the usual terminal-like interaction style with the C shell. VALET utilizes a command line interface although other researchers have demonstrated that a graphical, point-and-click user interface for the shell can yield significant improvements in users' abilities to effectively use a UNIX system. (This was described in Section 2.4.) However, although a wholesale replacement of the C shell interface could be beneficial, especially to inexperienced users, the focus of the research embodied in VALET is to understand and demonstrate how command line interfaces may be improved through intelligent, context-based processing of commands. Preservation of the C shell's normal command line interface may in fact benefit experienced users of the shell who are already proficient with the interface and who make frequent use of the special features of the shell's command language — file name patterns (*globs*), command and file name completion, and command history, for example.

This alludes to a second way in which VALET is transparent: namely, that VALET does not attempt to change the command language of the C shell, even though that language is far from ideal, both in terms of usability and in terms of conduciveness to intelligent analysis. Most significantly, as described in Section 1.3.1, almost all of the C shell's input language is determined and implemented by programs other than the shell itself. The names of most shell commands are simply the names of the other programs that the shell invokes — for example, `biff`, `awk`, and `grep`. These program names are often inexplicable because the various authors of these programs never thought to follow a consistent naming scheme. Furthermore, although the C shell invokes other programs it does not determine the syntax or semantics of other programs' command line arguments. Each program incorporates its own parser for command line arguments, and these parsers can vary widely from program to program. In short, because the shell's language has always been determined by programs beyond the shell's control, the shell's language was evolved rather than designed. From a user's viewpoint this means that the shell's input language is full of inconsistencies: Different programs require different command line syntaxes. From VALET's viewpoint this "distributed" language design is problematic because it means that the shell does not control — in fact, has almost no knowledge of — its own input language. Even worse, the shell has no programmatic way to gain knowledge about its command set. This means that an intelligent interface such as VALET must incorporate a priori knowledge about the programs that users may invoke. Because there are literally hundreds of programs in a modern UNIX system and because new programs are created almost continuously, it is impossible for an intelligent UNIX shell interface to have complete, built-in, detailed knowledge of all the commands that a user might invoke. Fortunately, however, people use only a very small number of commands with any frequency [12].

The C shell language has other features that hinder intelligent parsing of users' input commands. As just described, the set of available commands is not under the shell's control. Unfortunately this is not the only uncontrollable domain to which shell commands refer: The UNIX file system is another such domain. Many commands expect to receive file names as arguments, so VALET must maintain a detailed model of its host's file system. Moreover, because UNIX file systems change frequently, VALET must continually reexamine its host's file system in order to keep its internal model current. This continual rescanning is very time consuming — and even then, because it is not possible for a UNIX process to ask to be informed of all changes to a file system, VALET's file system model is always slightly out of date.

Finally, the C shell command language has lexical and syntactic features that impede intelligent input processing. Because the language is generally terse — the names of commands and command options are generally very short — there is often little lexical context that an intelligent interface can use in order to correct users' input mistakes. Fortunately, as described in Section 1.3.2, VALET can draw upon other kinds of context in order to detect and correct errors in terse inputs. Other difficulties arise from the complexity of the language itself. The shell language provides several kinds of syntactic shortcuts — including shell variables, file name patterns, and user-definable command aliases — and in addition provides syntax for directing the inputs and outputs of commands to files and to other commands through "pipelines." There are special syntaxes for "quoting" arguments in various ways. Handling all of these mechanisms requires a sophisticated parser. Ideally an intelligent command line interface to the shell would understand all of these syntactic features; VALET, however, understands only a subset of them for reasons described in Section 3.4.1.

In summary, the standard C shell language is a difficult language to parse intelligently. Some of the language obstacles are due to the shell's special role as a program that invokes other programs, and other obstacles arise from the shell's own syntactic features. Despite all of these problems with the standard C shell language, however, the VALET interface is designed to provide transparent access to the normal shell language. Although the standard shell language is difficult in many ways, it would have taken an incredible effort to replace this language with an all-new (and presumably, easier to use) command language. This new language could not possibly have extended to all of the hundreds of already available UNIX commands, at least not without reverting to the syntax of the existing shell language, and in any case a new language would likely not be acceptable to people who are already proficient with the existing shell language. Ultimately, because the motivation behind VALET was not to create a new shell language but rather to understand how intelligent command processing could benefit command line interfaces, VALET was designed to act as a transparent agent that parses the regular UNIX C shell command language.

A third and final aspect of VALET's transparent nature is that VALET does not attempt to parse, process, or augment the output from any of the UNIX programs run by the shell. Some of this output, particularly error messages, could provide important contextual information to an intelligent interface such as VALET. Certainly, the output messages of many programs could be improved or even tailored to the experience levels of individual users. However, the task of interpreting arbitrary program output is very large and complicated and is well beyond the scope and purpose of VALET. VALET is therefore transparent in the sense that it does not filter or interpret the output of any program (except that VALET must locate prompts from the shell for reasons detailed in Section 3.3.1). VALET is constrained to interpreting its users' input commands, based on contextual information derived from sources other than programs' output.

3.1.3 Summary of Features and Limitations

To summarize, the goal of VALET is to provide an "intelligent" interface to the UNIX C shell. VALET meets this goal by analyzing the shell commands that its users enter and by detecting errors in those commands. VALET maintains detailed knowledge bases that describe the context in which commands are given to the shell, and this knowledge includes:

- information about the complete set of shell commands available to each user, with a few minor exceptions described in Section 3.4.4;
- detailed knowledge (e.g., descriptions of command line syntax) of the most commonly used UNIX commands;
- a detailed, continually updated model of the UNIX host's file system;
- information about the state of the current shell session (e.g., the shell's current directory);
- both domain-independent and domain-dependent heuristics for correcting the components of faulty shell commands; and
- for each user, information about that user's interaction history, including the sets of commands and files that have been referenced — both referenced recently and referenced ever by that user.

Although much of VALET's knowledge base is static and shared by all of VALET's users, VALET also maintains additional context that is specific to individual users. VALET records a unique interaction profile for each user as just described, and each user's profile is preserved between sessions with VALET. These user-tailored profiles greatly improve VALET's ability to correct its users' shell commands.

VALET attempts to correct the input errors that computer users make most frequently: typographical errors, misspellings, incorrect file references, and other minor slips. VALET is also able to interpret many user-chosen abbreviations. When VALET recognizes an erroneous input command it attempts to correct that command; if VALET can determine a reasonable correction, it presents the revised command to the user for confirmation. Otherwise, when VALET cannot suggest a reasonable correction for a faulty input, VALET simply describes the error to the user and, depending on the error, presents the original command for editing by the user. In either case, VALET prevents detected erroneous commands from being received by the underlying UNIX shell.

VALET is an intelligent mediator that interprets communication from the user to the UNIX C shell. This mediation is largely transparent, however, meaning that VALET preserves almost all of the familiar UNIX C shell interface. As previously discussed in Section 3.1.2 VALET does not change the established textual, command-line interface style of the shell except as necessary to correct users' input errors. Similarly, VALET does not attempt to change the command language of the shell, nor does it attempt to analyze or augment the output of any program invoked by the shell. For these reasons VALET is most useful to people who are already familiar with the UNIX C shell and its command language. Although VALET's capabilities can certainly be useful to novice users, such users might benefit more from a graphical shell interface or a more intuitive command language, for example, than they would from VALET's ability to intelligently process commands in the existing shell language. Similarly, VALET is not an instructional tool. Although VALET can correct and explain certain kinds of errors, VALET does not try to teach its users about UNIX.

Finally, VALET was designed to be an exploratory vehicle for user interface research, intended to demonstrate the usefulness of intelligent, context-based, error-tolerant user interfaces. Although it is hoped that the ideas embodied in VALET will become widespread, it was never intended for the initial implementation of VALET to be a "production quality" or widely used interface to the UNIX shell. Rather, the current implementation of VALET was designed to be flexible and to support rapid prototyping.

3.2 An Overview of the Implementation

As illustrated in Figure 3.1, VALET is implemented as a collection of several separate but communicating UNIX processes. Although this may seem complicated, almost all of this complexity is hidden from VALET's users; from a user's perspective VALET is a unified intelligent shell. Internally, a standard, unmodified UNIX C shell process is

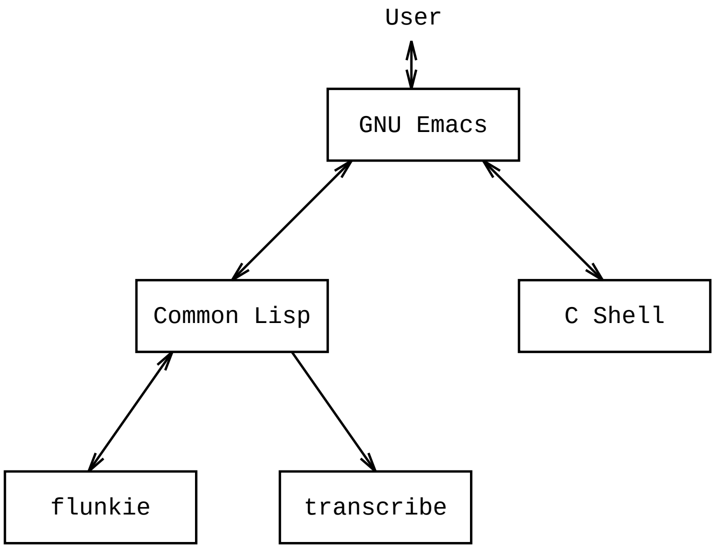

Figure 3.1. Cooperating Processes Within VALET

encapsulated within a framework that allows VALET to control the user interface of that shell process. VALET provides a terminal-like window for the shell, and output from the shell is presented immediately to the user. However, although shell output is sent directly to the user, user input is not sent directly to the shell. Rather, VALET intercepts and analyzes its users' commands. VALET can either pass these commands along to the shell or it can take other action — in particular, it may offer corrections to the user. VALET can display messages to the user in such a way that they appear to come from the shell itself. In this way, by carefully insinuating itself between the user and the UNIX shell, VALET creates the illusion that the user is interacting with a single, unified intelligent shell process.

At the top of VALET's process hierarchy, a GNU Emacs [20, 30] process provides both the actual user interface and the "glue" that connects the UNIX C shell process with the processes that intelligently interpret users' shell commands. GNU Emacs is a highly extensible text editing program and among its many features is the ability to run other programs. VALET makes extensive use of this feature. When a person begins a session with VALET, GNU Emacs creates a new C shell process. This shell communicates with its user through an associated Emacs text buffer; output from the shell and from programs run by the shell appears in this buffer. GNU Emacs displays the contents of this buffer (or a portion thereof) in a terminal-like window on the user's computer display, thereby providing the user with the regular textual interface to the UNIX shell. Additional special features of the GNU Emacs interface to the shell are described in Section 3.3.1.

The person using VALET gives commands to a shell by typing them in the shell buffer provided by GNU Emacs. However, as previously described these commands are not sent directly to the shell process. GNU Emacs instead sends the user's input to a second process, a Common Lisp [31] process, for intelligent analysis. This Common Lisp process was silently started by Emacs at the same time that Emacs started the shell. Unlike the VALET shell process, however, the Common Lisp process is hidden from VALET's user. The Common Lisp process communicates with GNU Emacs through an Emacs buffer, separate from the shell's buffer, and the Common Lisp's buffer is never normally displayed to the user.

This Common Lisp process is at the heart of VALET. It is this process that implements the intelligent parsing of users' input commands and determines how those commands should be answered. As just described, GNU Emacs sends users' shell commands to the Common Lisp process for interpretation. The Common Lisp process in turn responds to GNU Emacs with special instructions. Depending on the situation, the Common Lisp may tell Emacs to send the user's command to the shell, or it may tell Emacs to insert a special message into the shell's buffer — making it appear as if the message had been output by the underlying shell itself, not by VALET's Common Lisp process.

Two additional processes also communicate with VALET's Common Lisp. Because the Common Lisp programming language does not provide sufficient means for examining the UNIX file system, VALET uses a separate program called `flunkie` for this task. The Common Lisp process starts and communicates with `flunkie`, written in the C language, in order to get up-to-date information about the host's file system. In addition, a program named `transcribe` allows VALET to maintain secure transcripts of its users' sessions, detailing users' inputs and VALET's responses to those inputs. As described in Section 4.2 these transcript files provide data for evaluating the usefulness of the interface. Although transcript files could be written by VALET's Common Lisp process itself, a separate `transcribe` process is required in order to ensure that the session transcript files are stored securely, in a way that prevents unauthorized access to the data.

The decision to implement VALET as a collection of cooperating processes was made because this architecture provides maximal design flexibility and support for rapid prototyping — important features for an experimental system such as VALET. Some of the ways in which VALET benefited from its division into multiple processes are these:

- Most significantly, the division of the system into separate processes made it possible for VALET's intelligent command parser to be implemented in Common Lisp rather than in C, the implementation language of the UNIX C shell. Common Lisp, because it provides sophisticated facilities for symbolic programming, interactive program development, and experimentation, was an excellent implementation language for VALET's intelligent processing engine. Although a C language implementation could perhaps have been directly integrated with the C shell program, it would have been much more difficult to design, implement, and modify VALET's intelligent command interpreter in C.

- Once it was decided to implement the core of VALET in Common Lisp, it became necessary to create a bridge between the Common Lisp part of VALET and the C shell. GNU Emacs, the programmable editor, was the obvious choice. Not only did it provide the means to integrate the Common Lisp and shell processes but it also provided the ability to communicate with the user. GNU Emacs presents a terminal-like display to the user of the VALET system while in addition acting as the conduit between the Common Lisp and shell processes. It was possible to create this interface quickly and change it easily because GNU Emacs provides its own built-in programming language, Emacs Lisp [20].

- A final benefit of VALET's division into several processes is that it was easy to develop and test the individual components separately. Each part of the system could be tested interactively in isolation from the other parts.

The architecture of the VALET system also posed significant difficulties. Certainly, the division of the system into separate processes makes the interface slower than it would have been had VALET been implemented within a single UNIX process (presumably in the C programming language). In order to alleviate this problem the critical Common Lisp sections of VALET were carefully written to be very fast, and in general, VALET can respond to users' inputs with acceptable speed. More serious obstacles, however, also arose from the separation of the C shell and the rest of VALET's components. Two such problems are the following:

1. Because the C shell language contains so many features (e.g., command aliases, history references, and variable substitutions) it is difficult or impossible for an external command parser to interpret certain kinds of shell commands. In order to analyze commands that contain variable references, for example, a shell command parser must have knowledge of all the shell's variables and their respective values. This kind of information — information about the shell's internal state — can be very difficult or impossible to consult from outside the shell process itself. For this reason, VALET's shell command parser, which is implemented in a Common Lisp process external to the shell, does not recognize certain constructs within shell commands. The restrictions are explained in Sections 3.4.1 and 3.4.4.

2. The recognition of input shell commands, as opposed to other kinds of input, is based on heuristics. GNU Emacs has limited information about the state of the communication channel between itself and the shell. When the shell runs an interactive program, that new program takes control of the shell's communication channel with GNU Emacs. This happens without Emacs' knowledge, and therefore it is impossible for Emacs to unequivocally determine which process will receive an input sent along the shell's channel; it may be the shell itself or it may be another program run by the shell. This lack of knowledge is problematic because VALET is designed only to interpret shell commands, and in order for VALET to work properly GNU Emacs must direct shell commands to VALET's Common Lisp process while allowing inputs to other programs to pass through unimpeded. VALET's heuristic solution to this problem is described in Section 3.3.1.

Fortunately, however, these and other similar technical problems have little practical importance for VALET. Although the division between the C shell process and VALET's other components prevents the intelligent interface from accessing certain information about the shell, these limits are acceptable for an experimental, prototype system such as VALET. The problems caused by these technical difficulties amount to minor nuisances that are tangential to VALET's real purpose.

3.3 The GNU Emacs Components

As illustrated in Figure 3.2, GNU Emacs [20, 30] provides the actual interface between the VALET user and the intelligent UNIX shell. To start a session with VALET, a

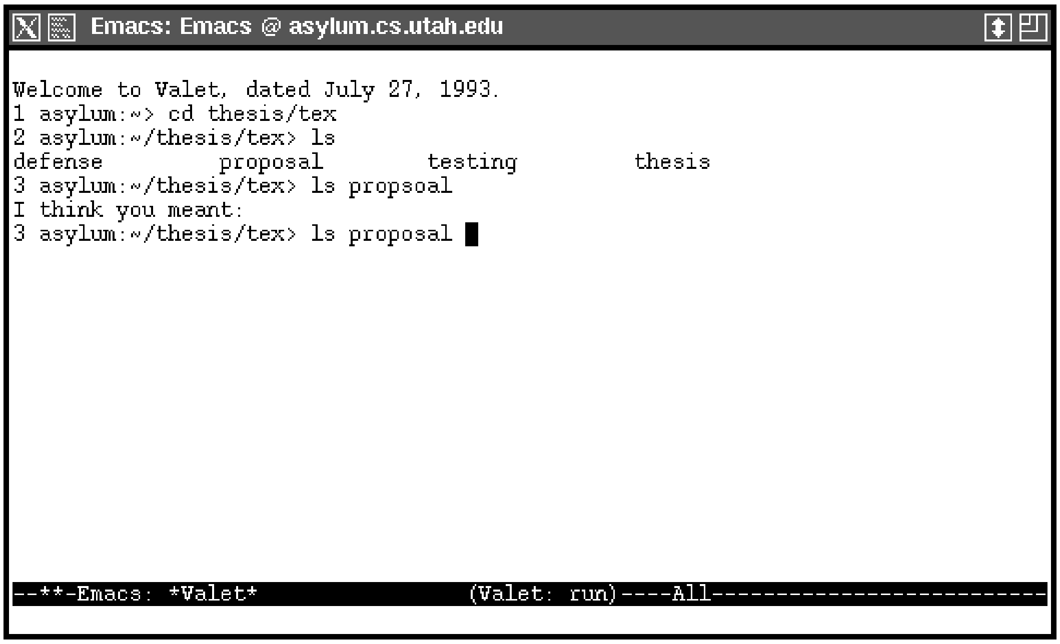

Figure 3.2. Appearance of the VALET Interface Within GNU Emacs

person first starts the GNU Emacs editor and issues the following command to Emacs: "`M-x valet`".[1] The VALET interface then appears in an Emacs window like the one shown in Figure 3.2. VALET introduces itself and presents the initial input prompt from the C shell.

VALET can actually provide intelligent interfaces for several independent C shell processes, all running simultaneously within a single VALET session. A single GNU Emacs process manages all of these shells. Each shell process is associated with a unique Emacs buffer, and each buffer appears in a separate Emacs window. (No matter how many shells are run, however, VALET creates only a single Common Lisp process to interpret its user's input shell commands.) To start each VALET shell process after the first, a person gives the command "`C-u M-x valet`"[2] to Emacs. Each invocation of this command starts a new VALET shell process running in a new Emacs window.

---

[1] "`M-x`" is the GNU Emacs notation for the character called `Meta-x`. An Emacs user enters a `M-x` character by depressing his or her keyboard's `Meta` key and then, while holding the `Meta` key down, pressing the `X` key.

[2] "`C-u`" is the GNU Emacs notation for the character `Control-u`.

3.3.1 Communication with the C Shells

Once a VALET shell process is running, a person can enter commands to that shell by simply typing them in that shell's buffer and then hitting the `Return` key, as if the user and shell were communicating through an ordinary terminal window. GNU Emacs does not simply act as a terminal, however. The most important difference is that GNU Emacs silently redirects users' shell commands to VALET's Common Lisp process for intelligent analysis; this treatment was previously described in Section 3.2. A second difference between a terminal interface and GNU Emacs is that Emacs enhances VALET's shell buffers with special features for interaction with the shells. For example, the VALET interface allows users to edit their input commands to a shell before those commands are submitted for execution. (Most of these editing capabilities are simply those that GNU Emacs itself provides for editing all kinds of text.) The VALET interface also defines special keystrokes for recalling commands that the user previously typed to the VALET shells. For instance, when a VALET shell buffer is selected, the key `M-p` recalls the user's immediately previous input command, inserting it after the current shell prompt as if the user had just typed the entire command. The user can submit this just-recalled command to the shell by hitting the `Return` key, or the user can edit the recalled command, or the user can press `M-p` a second time. A second press of `M-p` causes the just-recalled command to be replaced with the user's second most recent command. In general, `M-p` can be repeated as often as required in order to recall increasingly distant commands from the user's input history. Other keys iterate through a shell buffer's input history in different ways; for example, `M-s` recalls a previously entered command that starts with a particular prefix.

VALET's command editing and recall features are similar to those provided by the `tcsh` shell [27] with the important difference that VALET keeps two separate command histories for each shell process: one that holds inputs to the shell and one that holds inputs to the other programs run by the shell. The separation of these histories is beneficial to VALET users because in general, inputs to the shell are not suitable as inputs to other programs and vice versa. The VALET keystrokes that recall previously entered inputs (e.g., `M-p` and `M-s`) automatically consult the history list that is appropriate to the current situation. If a VALET shell is waiting for input then that shell's "shell command history" is searched; otherwise that shell's "non-shell input history" is searched. In this way VALET makes it easier for users to recall relevant prior inputs.

The separation of users' inputs into two histories for each shell process requires VALET's GNU Emacs component to differentiate between shell commands and other inputs. Dual histories are convenient but not fundamental to VALET, so one might believe that the ability to distinguish between shell inputs and other inputs is superfluous to the intelligent shell interface. That is not true, however. In fact, the ability to distinguish these two kinds of inputs is *essential* to VALET. The Common Lisp core of VALET can intelligently interpret *only* C shell commands, and all of the user inputs received by the VALET's Common Lisp process are treated as if they were shell commands. If the Common Lisp process were to receive inputs that were not in fact intended for the shell, the resultant interference with other programs' inputs would severely aggravate VALET's users and effectively overshadow VALET's merits. For this reason the GNU Emacs component of VALET must distinguish shell inputs from other inputs and treat them differently.

VALET's GNU Emacs interface must direct its users' C shell commands to VALET's Common Lisp component for intelligent processing while allowing inputs intended for other programs (i.e., the programs run by the shell) to be sent directly to those programs. Unfortunately, GNU Emacs cannot unequivocally determine which inputs are shell commands and which inputs are not. GNU Emacs has limited information about the states of the communication channels between itself and the VALET shell processes. For each shell process there is a unique bidirectional communication channel that connects the shell with Emacs. Commands sent by Emacs through a shell's channel may be received by the associated shell process or they may be received by an interactive program that is being run by that shell; GNU Emacs cannot distinguish these cases with absolute certainty. Fortunately there is a reliable heuristic that allows VALET to overcome this problem. VALET's GNU Emacs component simply searches the text it receives from each shell's channel for the prompts for input issued by the respective C shell processes. Every time a user input is submitted in a shell's buffer (i.e., every time the user hits the `Return` key in a shell's buffer), Emacs immediately checks the state of the associated shell channel. If the output from the channel ends with a shell prompt then Emacs classifies the user's current input as a shell command — and therefore, Emacs immediately redirects the input to VALET's Common Lisp process. If, however, the output from the shell channel does not end with a shell prompt, Emacs classifies the user's current input as non-shell input — not to be interpreted — and sends it through the shell channel. The use of shell prompts to distinguish shell inputs from non-shell inputs is highly effective, although it is certainly not an ideal solution. This heuristic can be fooled easily. In addition, it prevents VALET users from "typing ahead" by entering shell commands before the corresponding shell prompts are issued,[3] and it must be customized for individual users because different C shell users specify different shell prompt formats. These shortcomings are annoying, but in most cases VALET's GNU Emacs interface can accurately distinguish its users' shell commands from other kinds of input.

Except for the complications caused by the need to distinguish shell commands from other inputs, communication between Emacs and VALET's shell processes is straightforward. When GNU Emacs receives output from a shell (or from a program run by a shell), Emacs appends the new output to the buffer associated with the shell. Emacs scans the newly received output for shell prompts and makes the shell's buffer visible in an Emacs window if necessary. In addition to handling the output from VALET's shell processes, Emacs also regulates the input to those processes. Users' inputs directed to programs other than VALET's shells are sent directly by Emacs to those programs. Users' commands to the shells, however, are directed first to VALET's Common Lisp process.

3.3.2 Communication with Common Lisp

Whenever a person begins a session with VALET, GNU Emacs starts not only a new C shell process but also a new Common Lisp [31] process. VALET presents the new shell to the user of the system, as shown in Figure 3.2, but the Common Lisp process is hidden from view. The purpose of this hidden process is to analyze users' commands to VALET's C shell process (or processes) and to direct the VALET interface accordingly. Although GNU Emacs manages the actual communication with VALET's C shell processes, the "intelligence" of the VALET interface is contained not within GNU Emacs but within the Common Lisp process. Emacs' task, therefore, is to coordinate the Common Lisp process and the C shell processes. Emacs informs Common Lisp of events that concern the VALET shell processes, and in reply, Common Lisp sends instructions to Emacs. Emacs accepts these instructions and acts upon the shell processes on behalf of the Common Lisp process. All of this communication between VALET's Emacs and Common Lisp components is internal to VALET, completely hidden from the view of VALET's users.

[3] A VALET user may type a shell command before the corresponding shell prompt has been issued, but he or she must not actually submit the command (by pressing the `Return` key) before the prompt appears in the appropriate VALET shell's Emacs buffer.

From a user's perspective the intelligence of the system appears to be contained within the C shell processes.

VALET's Emacs and Common Lisp processes communicate by exchanging messages. Messages sent by Emacs to Common Lisp are received by the Common Lisp process' interpreter (i.e., the Lisp's *read-eval-print loop* or *top-level loop*), so these messages are formatted as Lisp symbolic expressions. Table 3.1 summarizes the forms that are sent from Emacs to Common Lisp. The events that cause messages to be sent are the following:

Initialization of the interface. Immediately after GNU Emacs creates VALET's Common Lisp process, Emacs sends certain initialization commands to it. These forms tell Common Lisp to load the VALET Common Lisp code, initialize its data structures, load the current user's saved interaction history, start the `flunkie` and `transcribe` processes, and take other similar actions. Because these introductory forms are somewhat long and tedious they have been omitted from Table 3.1.

Creation of a shell process. Each time a new C shell process starts, Emacs informs the Common Lisp process of the event. The `valet-new-shell` message contains two arguments. The first, called *shell-number*, is an integer that Emacs associates with the new shell process. This number serves to uniquely identify this shell process in future messages between Emacs and Common Lisp. The second datum in the `valet-new-shell` message is a string containing the absolute pathname of the new shell's initial working directory.

Termination of a shell process. Just as Emacs reports the creation of new shells, Emacs also informs Common Lisp whenever a VALET shell process terminates.

Table 3.1. Summary of Messages Sent From GNU Emacs to Common Lisp

Event	**Message Sent from Emacs to Common Lisp**
Initialization	...
Creation of a shell	`(valet-new-shell` *shell-number directory*`)`
Termination of a shell	`(valet-dead-shell` *shell-number*`)`
User input to a shell	`(valet-user-input` *shell-number input index*`)`
User annotation	`(valet-wrong-correction` *shell-number* ...`)`
User annotation	`(valet-user-note` *notation*`)`
Termination of Emacs	`(valet-save)`

User input to a shell process. GNU Emacs submits input shell commands to the Common Lisp process by sending a `valet-user-input` message. Contained in the message, *shell-number* is the identifying number of the shell that received input, and *input* is a string containing the input. The final datum, *index*, is the index of this input in the shell command history list that Emacs maintains for the shell, as described in Section 3.3.1. (The *index* is currently unused by Common Lisp.)

User annotation. The GNU Emacs interface to VALET provides two ways for a user of the system to insert messages into his or her VALET session transcript. The first, `valet-wrong-correction`, allows a user to record that VALET made an erroneous attempt to correct the user's most recent shell command. The data in the `valet-wrong-correction` message include the shell number, the user's original input, VALET's suggested correction, and the correction that was desired by the user. The second kind of annotation, `valet-user-note`, permits a user to insert general comments into the transcript. GNU Emacs defines special keystrokes that allow users to make each kind of annotation. However, because the session transcript is maintained by Common Lisp (and its auxiliary `transcribe` process), Emacs must relay its users' comments to VALET's Common Lisp.

Termination of GNU Emacs. Finally, if the VALET user has attempted to terminate Emacs itself, then Emacs sends a `valet-save` message to the Common Lisp process. This allows VALET to save the user's session data as described in Section 3.4.6. Emacs does not actually terminate until the Common Lisp process indicates that the session data have been safely stored.

With the exception of the final message type, none of the above-listed messages requires Emacs to wait for a reply or acknowledgment from VALET's Common Lisp process. Emacs can perform other tasks if necessary until it receives instructions from Common Lisp.

Just as Emacs sends messages to Common Lisp, Common Lisp sends messages to Emacs. Emacs reads and evaluates the Lisp expressions that it receives from VALET's Common Lisp process, thereby carrying out the Common Lisp's commands.[4] Table 3.2

[4]Because Common Lisp's commands to Emacs are intermingled with other kinds of output from the Common Lisp process, Emacs does not attempt to evaluate *everything* that it receives from Common Lisp. Instead, VALET's Emacs code searches for specially marked expressions in the Common Lisp output and evaluates only those forms.

Table 3.2. Summary of Messages Sent From Common Lisp to GNU Emacs

Requested Action	**Message Sent from Common Lisp to Emacs**
Send input to a shell	`(valet-send-to-shell` *shell-number input*`)`
Output a shell message	`(valet-shell-display-message` *shell-number message*`)`
Reissue a shell prompt	`(valet-shell-reprompt` *shell-number* `&optional` *input*`)`
Change directory	`(valet-shell-cd` *shell-number directory*`)`
Set correction data	`(valet-shell-set-cc-info` *shell-number input data*`)`
Allow Emacs to exit	`(valet-lisp-saved)`

summarizes the instructions that VALET's Common Lisp process sends to Emacs. These instructions include:

Send input to a shell process. The first kind of direction tells GNU Emacs to send a specific string as input to a shell process. The data contained in this message are obvious.

Output a message from a shell. VALET's Common Lisp process can instruct GNU Emacs to insert a message in a shell's buffer, making the message appear to be output from the shell process itself. The ability to forge output from VALET's shell processes in this way is essential to the illusion created by VALET — namely, that the intelligence of the system is embodied in the shells themselves and not within an external Common Lisp.

Reissue a shell prompt. The Common Lisp process can also tell Emacs to forge a prompt from a shell process. This new prompt is identical to the last prompt issued by the shell. In addition, Common Lisp can specify that an *input* string should be inserted after the new prompt, as if it had just been typed by the VALET user. In general this *input* string will be VALET's corrected version of the user's previously entered command. With the corrected input already inserted in the shell's buffer, VALET's user can simply press the `Return` key in order to accept VALET's corrected version of the command.

Change a shell buffer's directory. When VALET's Common Lisp process determines that a shell process will change its working directory, Common Lisp tells Emacs to correspondingly change the directory that Emacs associates with the shell's buffer.

The argument *directory* is the absolute pathname of the new working directory of the indicated shell process. The ability for Emacs to track shell processes' directories is essentially a convenience for VALET's users.

Set shell command correction data. This message instructs Emacs to remember certain data about VALET's interpretation of the last command given to the indicated shell. This information is used by Emacs if the VALET user asks to annotate the session transcript and indicate that VALET's interpretation of the previous shell command was incorrect. Within the message, *input* is the user's previous input to the shell process and *data* is either the corrected command or the Lisp symbol `nil`, indicating that VALET accepted the user's input verbatim.

Allow Emacs to terminate. This final message is sent from Common Lisp to Emacs in order to tell Emacs that VALET has safely saved the data from the current session for future use. This knowledge allows GNU Emacs to terminate, thereby terminating all of the VALET interface processes.

As with all but one of Emacs' messages to Common Lisp, none of the messages sent from Common Lisp to Emacs require the Common Lisp process to wait for a response or acknowledgment. This allows the Common Lisp process to send several messages at one time (e.g., `valet-shell-display-message` followed immediately by `valet-shell-reprompt`) or to send a message to Emacs and then perform other computations.

In summary, the messages exchanged by VALET's Common Lisp and Emacs components allow these separate processes to cooperate in providing a sophisticated interface to VALET's C shell processes. When a VALET user enters a shell command, GNU Emacs silently redirects the input and sends a `valet-user-input` message containing the user's input to Common Lisp. The Common Lisp process in turn interprets the command and responds to Emacs with instructions — perhaps with a `valet-send-to-shell` message telling Emacs to send the user's (valid) command to the appropriate shell process. All of the transactions between Emacs and Common Lisp are normally hidden within the VALET system itself. However, because the parts of VALET are implemented as separate processes it is possible for the implementors of VALET to break the system apart and, for example, interactively communicate with the Common Lisp components of the system, even while the interface is operating.

3.4 The Common Lisp Components

The purpose of the Common Lisp process within the VALET interface is to "intelligently" parse and respond to users' shell commands. VALET's shell command parser, the knowledge bases that support that parser, and the heuristics that correct malformed commands are all contained within the Common Lisp process.

As previously detailed in Section 3.3.2, VALET's Common Lisp component receives messages from GNU Emacs describing the events that occur in the interface. Each message from Emacs is formatted as a Common Lisp symbolic expression and is received and executed by the Common Lisp's read-eval-print loop. The most common — and also most significant — messages sent from Emacs to Common Lisp are those that pass users' input shell commands to the Common Lisp process. The function that receives these messages, `valet-user-input`, relies on many different modules within the Common Lisp process in order to produce an appropriate response. These modules include:

- an input tokenizer that divides the command line into a list of separate words;
- an augmented transition network (ATN) parser with the necessary networks and action functions to interpret and, as necessary, correct the tokenized input;
- knowledge bases for several domains (e.g., the available shell commands and the UNIX host's file system) that support the ATN parser's actions; and
- provisions for preserving the knowledge bases and the user's interaction history between separate VALET sessions.

These components of the VALET system are described below.

3.4.1 The Input Tokenizer

The first step in VALET's interpretation of a shell command is to divide the user's input command line into its constituent lexemes — also called *tokens* — or informally, words. The division of a string into lexemes is called *lexical analysis* or *tokenization* and is generally the initial processing step for all programs that interpret some kind of input language. For many computer languages the process of lexical analysis is straightforward; however, because the C shell input language [17] defines several different and expressive abbreviation mechanisms, VALET's input tokenizer is actually quite complex. In fact, the implementation of the tokenizer contains over 1,000 lines of Common Lisp code. The lexical analysis of a C shell command requires several steps:

- First, the command line is divided into lexemes according to the C shell's input conventions.
- Next, tokens that contain braces ("`{`" and "`}`") are expanded; each token that contains a brace construction is replaced by one or more tokens that contain the expanded term.
- Third, abbreviations that refer to users' home directories ("~" and "~*user*") are interpreted and resolved.
- Finally, file name patterns (which contain any of the special characters "`*`", "`?`", "`[`", and "`]`") are expanded. Each token that contains a file name pattern is replaced by zero or more tokens that contain the set of matched file names.

The first stage of VALET's tokenizer divides the original shell command line, a string, into a list of `token` data structures; Figure 3.3 contains the definition of the `token` structure. Each `token` is described by several fields. The first, called `type`, is a symbol that identifies the syntactic role of the token. For instance, a token may represent a special construct such as a pipe or an I/O redirection, or it may simply be a command line argument. The second `token` slot, `input`, contains the string of input characters that

```
(defstruct (token ...)
  (type       nil  :type symbol         :read-only t) ;; Type of this token.

  (input      ""   :type simple-string  :read-only t) ;; Chars as input...
  (string     ""   :type simple-string  :read-only t) ;;   ...and as read.
  (quotes     nil  :type list           :read-only t) ;; Quoted char ranges.

  (obscure-p  nil  :type symbol         :read-only t) ;; Understood by Valet?
  (children   nil  :type list                      ) ;; Results of expansion.
  )
```

Figure 3.3. Definition of the `token` Structure

were parsed in order to produce this token.[5] This is separate from the `string` slot because the C shell input language includes various *quotation* constructs; these lexical forms called *quotes* do not represent input themselves but instead define the lexical properties of other input characters. The `input` slot of a token contains the quotation characters, if any, that appear in the user's original command line. The `string` and `quotes` slots within a `token`, on the other hand, describe the input token after quoted constructs within the original input have been interpreted.

The `obscure-p` slot is used to mark tokens that, although they are well-formed inputs, VALET cannot properly interpret. The current VALET system cannot handle several of the C shell's input abbreviation facilities; in particular, VALET cannot understand backquoted strings or variable references or command history references (i.e., "`$`" and "`!`" constructs). These kinds of inputs are difficult or impossible for VALET to interpret correctly because VALET's command interpreter is separate from the managed C shell processes and therefore has limited access to the internal states of those shells. VALET can and does, however, correctly process most of the abbreviation mechanisms that are not based on the shells' internal states. Most significantly, VALET can process file name patterns. In addition, because VALET's GNU Emacs component provides its own command history mechanisms, the effects of the tokenizer's inability to handle variable and history references are mitigated. (Nevertheless, as described in Chapter 4, users occasionally stumbled over VALET's various limitations.)

The final slot in each `token` structure, `children`, is used in the three input expansion steps listed previously. The initial stage of the tokenizer produces a list of uninterpreted `token` structures. The second stage, brace expansion, searches for tokens that contain brace ("`{}`") pairs, and when a token containing a brace construct is found, the construct is expanded and the resultant list of new tokens is stored in the `children` slot of the now-expanded `token`. For example, brace expansion of the lexeme "`program.{c,h}`" produces two new lexemes — "`program.c`" and "`program.h`" — that are stored in the `children` list of the original "`program.{c,h}`" `token` structure. Similarly, when home directory abbreviations and file name patterns are processed, the resultant file name tokens are stored in the `children` slots of the now-expanded parent tokens.

[5] The input that produced the token is saved principally so that it may be used if VALET later needs to formulate a corrected command. By saving the user's original input, VALET can create a new command that greatly resembles the user's original input.

The three input expansion steps are applied in separate stages. The initial brace expansion step is fairly straightforward because brace abbreviations do not refer to data other than that in the abbreviated lexemes themselves. The replacement of tilde constructs, however, requires the tokenizer to consult the external `flunkie` process (described in Section 3.4.5.2) in order to determine the appropriate users' home directories. File name pattern expansion is even more involved — the tokenizer must first compile the patterns into more manageable forms and then interact with VALET's file system knowledge base, which is described in Section 3.4.5. In fact, VALET's lexical analyzer dynamically compiles file name patterns into augmented transition networks and then uses the parser described in Section 3.4.2 to direct the actual file name searches within the file system knowledge base — quite a task!

Ultimately, the process of token expansion produces a token "tree." The initial tokenization step creates a list of `token` structures, and each of the subsequent token expansion stages applies only to the "leaves" of the current token tree — in other words, those tokens that have not already been expanded by a prior pass of the tokenizer. The leaves of the token tree are the ultimate product of VALET's lexical analyzer. When all of the analysis and expansion steps are complete, VALET's tokenizer creates and returns a list containing all of the fully expanded tokens. This list of `token` structures becomes the input for the next stage of VALET's shell command interpreter.

3.4.2 The Augmented Transition Network Parser

After VALET has translated its user's shell command into a list of `token` structures, the next task is to actually interpret — and if necessary, correct — the user's command. This task is performed by a second module within VALET's Common Lisp process: an *augmented transition network* (ATN) *parser* [4]. VALET's ATN parser is quite sophisticated and provides features essential to the intelligent handling of shell commands:

- The parser provides convenient and powerful syntaxes for defining transition networks and their constituent actions.
- The parser is fully backtracking and allows for "redoable" actions.
- When a user's shell command can be neither interpreted nor corrected, the parser can explain why.

Ultimately, VALET's ATN parser is the engine that powers the intelligent processing of shell commands.

3.4.2.1 Transition Networks

An augmented transition network parser is so called because the operation of the parser is directed by a set of *transition networks*.[6] These networks determine the actions that the parser may take and the choices that it may make in order to produce an interpreted result from its input. In this way the networks define the language accepted by the parser. VALET, of course, defines networks that allow the ATN parser to interpret and correct shell commands.

One may envision a transition network as a directed graph — a set of nodes connected by directional links that allow an ATN parser to "travel" from one location in the graph to another. There is a designated initial node of the network and a designated final node. Further, every link between nodes in the network is associated with an action that the parser must perform in order to move across the link. Given a transition network, then, the goal of an ATN parser is to traverse its assigned network by traveling from the initial network node to the final network node. By accomplishing this task the parser successfully interprets its input.

The parser, however, is restricted in its movement through the network. As just described, each link within the network is associated with a specific action that the parser must perform in order to travel along that link. An action might, for instance, direct the parser to move a token from its input list to its output list. Depending on the state of the parser when it comes to this action, the parser may or may not be able to perform the task (e.g., prior actions may have consumed all of the parser's input tokens). If the parser cannot carry out the required task then it cannot travel across the action's associated link. Because travel through a transition network is restricted by the actions that the parser must perform along the way, an ATN parser must in general search for a path that allows it to traverse its assigned network.

From any node in a transition network there may be any number of links, each with its own associated action, that lead to other nodes. When several links lead away from the parser's current position, the parser must make a choice: It must select one of the

[6] "Augmented" refers to the parser's ability to maintain extra state information during its operation, thereby allowing the parser to base its actions and decisions on context.

links and attempt to traverse it. Unfortunately, the parser cannot know which of the links (if any) it should cross in order to proceed toward its ultimate destination. This means that in attempting to traverse a network the parser is likely to make choices that lead to "dead ends" — points from which there are no possible paths to the parser's ultimate destination.

For this reason VALET's ATN parser is able to *backtrack*. VALET's parser is fully backtracking, meaning that it remembers every choice that it makes during its operation. If at any point the parser discovers that it cannot proceed from its current location, the parser retreats to the most recent point that provides an unexplored alternative. From that point the parser attempts to move forward along a previously ignored link. If that newly chosen link leads to another dead end, the parser again backtracks to the most recent point that provides yet another untested alternative. The process repeats as often as necessary. Eventually, through methodic exploration of the possibilities, the ATN parser will either discover a path to its ultimate destination (and thereby successfully parse its input), or it will exhaust all of the alternatives and thus discover that there is no path to its destination (and therefore, no way to parse its input).

Although it is easy to visualize transition networks as directed graphs, when describing a transition network to a computer it is generally more convenient to use a textual representation of the network. VALET's transition networks are written in the Lisp-like notation shown in Table 3.3. (Internally, the notation is compiled into data structures that resemble directed graphs.) Each of the expressions in Table 3.3 defines a transition network. Some of the expressions contain other network definitions, so the notation is recursive.

Table 3.3. The Grammar Used to Define Transition Networks

Expression	**Meaning**
(*action-name arg-1 arg-2 ...*)	Perform the specified action.
(`seq` *net-1 net-2 ...*)	Traverse the sequence of networks.
(`or` *net-1 net-2 ...*)	A choice: *net-1* or *net-2* or ...
(`opt` *net*)	Traverse *net* zero or one time.
(`opt*` *net*)	Traverse *net* zero or more times.
(`one+` *net*)	Traverse *net* one or more times.
(`parse` *net-name*)	Traverse network named *net-name*.
(`parse (parse-state-var` *var*))	Traverse network stored in *var*.

The basic network definition "`(`*action-name* ...`)`" simply instructs the ATN parser to perform the specified action. In terms of the graph metaphor, this form defines the action associated with a directed link between two nodes in a transition network. Within the form, *action-name* is a symbol that names the action to be performed. Additional data for the action, if any, appear after the *action-name* as if they were arguments to a Lisp function. (However, unlike an ordinary Lisp function invocation, the arguments to a parser action are never evaluated.) As detailed later in Section 3.4.2.2, actions are essentially special Lisp functions that operate on the parser's current state. An action may attempt to consume some of the parser's unprocessed input, produce output, set or change the parser's internal variables, or do all of these things. An action may *succeed*, meaning that the parser may continue (i.e., move across the action's link), or an action may *fail*, meaning that the parser must retreat.

The form "`(seq` *net-1* *net-2* ...`)`" defines a network that is a sequence of other networks. In order for the parser to traverse this sequence, it must travel first through *net-1*, then through *net-2*, and then through all of the other specified networks. There may be any number of networks within the `seq` form. It is important to realize that each of these networks — *net-1*, *net-2*, and so on — may be described by *any* of the forms presented in Table 3.3. In other words, the components of a `seq` may be simple actions or they may be more complex network descriptions. This is the case for all of the expressions that contain other network descriptions: `seq`, `or`, `opt`, `opt*`, and `one+`.

The form "`(or` *net-1* *net-2* ...`)`" describes a choice: The parser may traverse *net-1*, it may traverse *net-2*, or it may traverse any one of the other networks within the `or` form. In terms of the graph metaphor, `or` defines a set of separate paths that all start from one node and all end in another. The parser will first attempt to traverse the network described by *net-1*. If the parser cannot find a path through *net-1*, or if the parser later on discovers that by traversing *net-1* it cannot reach its ultimate destination, the parser will backtrack to the `or` form and try the next alternative, *net-2*. The process of choosing and backtracking continues until either (1) the parser discovers a path to its ultimate destination, or (2) all of the alternatives in the `or` form have failed. In the latter case the parser will backtrack past the `or` form to a prior choice point.

The `opt`, `opt*`, and `one+` forms describe other kinds of choices to the ATN parser. The form "`(opt` *net*`)`" indicates that the parser may choose either to traverse or to ignore the enclosed *net*; in other words, the *net* is optional and may be traversed either zero times or one time. The form "`(opt*` *net*`)`" allows the parser to traverse the indicated *net* any number of times, including zero times, before moving on. Similarly, the form "`(one+` *net*`)`" tells the parser that it may traverse the enclosed *net* one or more times. As previously described, VALET's ATN parser is completely backtracking, so it remembers and can reconsider the choices it makes while traversing the structures described by `opt`, `opt*`, and `one+`. The parser may backtrack and decide to traverse a previously ignored optional network, for example. The `opt*` and `one+` forms describe loops in a transition network and the parser may backtrack in order to change the number of times it travels around each loop.

Finally, the `parse` forms allow transition networks to refer to other transition networks. The expression "`(parse` *net-name*`)`" is a special action that tells the parser that in order to cross the associated link, it must first completely traverse the transition network named by *net-name*. (The mechanism for naming networks is described below.) In a way, this is like a subroutine call to the named network. The second form of the `parse` action tells the ATN parser than in order to cross the associated link it must first traverse the transition network stored in the parser variable named *var*. (Transition networks are represented as first-class Lisp objects. Parser variables are described in Section 3.4.2.2.) If the first form of `parse` is like an invocation of a named subroutine, then the second form of `parse` is like an invocation of an anonymous subroutine.

Figure 3.4 illustrates how the notation just described is used to create VALET's transition networks. The macro `define-net` creates a named network; the new network can then be referenced in "`(parse` *net-name*`)`" actions. The function `make-net`, not shown in the figure, creates an anonymous network. The `make-net` function compiles a network description into a Lisp object and then returns that object. This compiled network can then be stored in a parser variable and later referenced through an appropriate "`(parse (parse-state-var` *var*`))`" action.

3.4.2.2 Transition Network Actions

The nodes and links within a transition network provide structure, but it is the actions associated with the links that specify the actual steps of the parsing task. As previously described, every link within a transition network is labeled with an action that must be performed by the parser in order for the parser to "travel" over the link. In VALET these actions, except for `parse`, are implemented as special Common Lisp functions.

Fundamentally, the purpose of an action is to examine or modify the ATN parser's

```
;;; Parse a simple command along with I/O redirection and ‘&’.

(define-net command
  (seq (set-semantics nil)
       (parse simple-command)
       (save-semantics)
       (no-more-command-arguments)
       (no-pretend-equal-var)
       (opt* (or (parse input-redirection)
                 (parse output-redirection))
             )
       (opt (parse background))
       (empty-input)
       ))

;;; Parse a command name and arguments.  The ‘simple-command-name’ action
;;; accepts the command name, locates the network for parsing that command’s
;;; arguments, and stores that network in ‘$$simple-command-net’.

(define-net simple-command
  (seq (simple-command-name)
       (parse (parse-state-var $$simple-command-net))
       ))

;;; Parse ‘< file’ or ‘<< word’.

(define-net input-redirection
  (seq (or (seq (literal-token :input-from-file)
                (file-name (:type :not-directory) (:mode :readable))
                )
           (seq (literal-token :input-from-stdin)
                (eat-arguments :min 1 :max 1 :name "word")
                )
           )
       (mark-as-parsed :input-redirection)
       ))

;;; Parse ‘> file’ or ‘>> file’ or ‘>& file’ or ‘>>& file’.

(define-net output-redirection
  (seq (literal-token :output-to-file :append-output-to-file)
       (opt (literal-token :ampersand))
       (file-name (:type :not-directory :imaginary) (:mode :writable))
       (mark-as-parsed :output-redirection)
       ))

;;; Parse ‘&’ for background processes.

(define-net background
  (seq (literal-token :ampersand)
       (mark-as-parsed :background)))
```

Figure 3.4. Example Transition Network Definitions

current state. This state includes the current *input* to the parser, the current *output* of the parser, and a set of *variables* maintained by the parser. For VALET, the input to the parser is a list of `token` structures (as described in Section 3.4.1) and the output of the parser is a partial parse tree that describes the portion of the input that has been processed. An action may modify both the parser's current input (e.g., "consume" an input token by removing it from the input token list) and output (e.g., produce the parsed version of that token). In addition, an action may also examine or set one or more of the parser's variables. VALET's parser maintains its own set of variables for use by its actions. Action functions may examine or set the values of these variables, for example, in order to remember the properties associated with tokens that have already been parsed.

When an action is able to perform its task — for example, modify the parser's state in a particular way or verify that a certain condition is true — the action *succeeds*. This allows the ATN parser to travel over the link associated with the action as previously described. On the other hand, an action *fails* when it cannot perform its task. This prevents the parser from crossing the action's associated link and forces the parser to search for another path though its network — in other words, to backtrack. When the ATN parser is forced to backtrack to a prior choice point, the parser restores its state — input, output, and variables — to the state remembered along with that choice point. This remembered state is the state the parser had before it moved forward from the choice point. In effect, by restoring its state the parser "undoes" all of the actions that were executed since the parser left the choice point. With its state thus restored the parser may choose a previously unattempted course of action.

A successful action allows the ATN parser to proceed whereas a failed action forces the parser to backtrack and consider other actions. Sometimes, however, the possible outcomes of an action are more complex than a simple choice between success and failure. Sometimes an action can succeed in several different ways, each way resulting in a different parser state. This situation might occur, for example, in an action designed to correct spelling. Such an action might read a token from the parser's current input, apply a spelling correction algorithm to that token, and then write the corrected token to the parser's output. If the correction algorithm were to suggest several different corrections of the original token, then the action could succeed by choosing any one of the possibilities. However, it may be that only one of the possibilities will allow the parser to complete its entire parsing task, and the action may not have enough information to select this one "right" correction from the list of alternatives.

Situations like this are typically resolved by dividing the problematic action into several separate actions. For example, the above-described action could be rewritten as two actions: The first would consume the input token and produce a list of alternative corrections, and the second would select one of the untested corrections and move it to the parser's output. Arranged in an appropriate network and combined with the parser's ability to backtrack, these new actions will eventually choose the "right" correction.

The transformation of one action into several networked actions is an effective technique, but VALET's ATN parser provides a second and sometimes more convenient option: *redoable* actions. A redoable action is one that can succeed in several different ways, producing several different parser states as just described. VALET's ATN parser recognizes redoable actions and treats them specially. Whenever VALET's parser successfully executes a redoable action it associates a special choice point with that action. Later, if the parser needs to backtrack, it may return to this choice point and reexecute the action. When a redoable action is reexecuted, the state of the parser (input, output, and variables) is the state that was produced by the previous invocation of the action. This allows information to be communicated between separate invocations of the action. As one would expect, when a redoable action is reexecuted, it will either succeed or fail. If it succeeds, then the parser updates the choice point associated with the redoable action and then moves forward. Otherwise, if the redoable action fails, the parser backtracks to the next most recent choice point. In summary, a redoable action is an action that may be executed repeatedly until it fails. In order to cross a transition network link labeled with a redoable action, VALET's ATN parser may invoke the redoable action's function several times in order to produce several different parser states.

As previously mentioned, the actions associated with VALET's transition network links are implemented by Common Lisp functions that operate upon the parser's state. When the parser attempts to perform an action, it invokes the function that implements that action. Because all action functions share a common structure and must communicate with the ATN parser in certain stylized ways, VALET's action functions are defined through the use of a special macro, `define-action`. Figure 3.5 contains some examples of VALET's action definitions.[7]

[7]The definitions shown in Figure 3.5 are actually slightly simplified versions of the definitions that VALET uses.

```
;;; Accept a token of a specified type.  If found, move it to the output.

(define-action literal-token
  :args (&rest accept-token-types)
  :body (let* ((input (get-input))
               (next-token (first input)))
          (when (null next-token)
            ;; There is no next token to accept.
            (fail))
          (let ((token-type (token-type next-token)))
            (if (member token-type accept-token-types)
                ;; Move ‘next-token’ to the output and succeed.
                (set-in/out (cdr input)
                            (cons `(literal-token ,next-token) (get-output)))
                ;; Otherwise, fail.
                (fail))
            ))
  )

;;; Remember that a particular kind of command component has been parsed.  If
;;; the parser tries to accept a second instance of the component, fail.

(define-action mark-as-parsed
  :args (component)
  :vars ($$parsed-components)
  :body (if (member component $$parsed-components)
            (fail)
            (setf $$parsed-components
                  (cons component $$parsed-components)))
  :fail (case component
          (:input-redirection  "there was more than one input redirection")
          (:output-redirection "there was more than one output redirection")
          (:background         "there were too many &'s")
          (T                   (format nil "there were too many ~S" component))
          )
  )

;;; Check the parser variable ‘$pretend-equal-var’.  If it is set, fail.

(define-action no-pretend-equal-var
  :worth 0
  :vars  ($pretend-equal-var)
  :body  (when $pretend-equal-var
           (fail))
  :fail  (third $pretend-equal-var) ;; Return the saved explanation.
  )
```

Figure 3.5. Example Transition Network Action Definitions

Figure 3.5 shows that every action definition is a collection of parts. Most obviously, every action has a name. Some actions expect to receive data from the parser each time they are invoked; these argument values are specified within the transition network links as described in Section 3.4.2.1. The definition of an action that receives arguments from the parser must include an `:args` list that describes those arguments. In Lisp terms, an `:args` list is the *lambda-list* for its action. The `:body` of an action is the Common Lisp code that implements that action. The code within the `:body` of an action may refer to the arguments passed to that action by the parser, as specified by the action's `:args` list. In addition, the `:body` of an action may refer to the parser variables listed in the action's `:vars` list as if they were ordinary Lisp variables.[8] By convention the names of parser variables begin with "`$`" but this is not required. Finally, the code within the `:body` of an action is simplified by various macros — `get-input`, `get-output`, `set-in/out`, and others — that the parser defines. These macros provide convenient access to the parser's current state and allow actions to perform other common tasks.

As previously explained, an attempt to perform an action will either succeed or fail. This result is determined by macros. The macro `succeed` signals to the parser that an action has succeeded. When invoked within an action's `:body`, `succeed` halts execution of the action's `:body` form and immediately returns control to the parser. The Lisp value returned to the parser indicates that the action succeeded. Conversely, the macro `fail` signals to the parser that an action has failed. The `fail` macro halts execution of the current action body, returns control to the parser, and informs the parser that the just-attempted action was unsuccessful. If the parser is able to execute the entire body of an action — that is, if control reaches the end of an action's `:body` form without invoking either `succeed` or `fail` — then the action is deemed to have succeeded.

Redoable actions, which may succeed in several different ways, contain a `:redo` form in addition to a `:body` form. (The presence of a `:redo` form in an action definition indicates that the action is redoable. Because none of the actions shown in Figure 3.5 are redoable, none of them contain `:redo` expressions.) The `:body` and `:redo` forms are similar except that they are evaluated at different times. The first time a redoable action is attempted, the parser executes the action's `:body`, as it does for all actions. However, the second

---

[8]An action's `:vars` list allows for syntactically simple access to the named parser variables. An action can refer to parser variables that are not contained in its `:vars` list, but the syntax for doing so is less elegant.

and subsequent times that a redoable action is attempted, the parser invokes the action's `:redo` form. As previously described, when a redoable action is reexecuted, the state of the parser is the state that was produced by the previous invocation of that action. The purpose of the `:redo` form, then, is to modify that state, which was produced by the previous execution of either the action's `:body` or `:redo` forms. Like the `:body`, the `:redo` form may succeed or fail.

It is possible to define local functions within an action by specifying a `:funs` list, although none of the actions defined in Figure 3.5 do so. The final two parts of an action definition, `:worth` and `:fail`, are used when the parser is unable to interpret its input successfully and completely — in other words, when the parser needs to explain why its input failed to parse.

3.4.2.3 Explanation of Parsing Failures

Given a transition network, the goal of VALET's ATN parser is to traverse that network by traveling from the network's initial node to its final node. In the process of traversing the network, the ATN parser interprets its input and constructs its output. If the parser reaches the final node of its network, then it is deemed to have *succeeded* — successfully parsed its input. The parser returns a Lisp structure that describes the parser's final state, including the parse tree generated from the accepted input. If, however, the ATN parser cannot find any path to the final node of its network, then the parser is deemed to have *failed*, and when the parser fails it simply returns `nil`. This result is only minimally informative but fortunately the parser remembers *why* it failed and can help VALET construct an appropriate explanation.

During the course of its operation the ATN parser often runs into "dead ends" from which it must backtrack. Even when the parser is ultimately successful, it may encounter and retreat from many dead ends before it finally discovers a path through its assigned network. The fact that the parser encounters dead ends before it succeeds is due to exploratory nature of the parser itself; in particular, these dead ends do not indicate that the parser's input is malformed. Obviously, in order for the parser to succeed (i.e., interpret all of its input), the parser's input must be well-formed according to the language described by the traversed networks. The situation is somewhat different, however, when the parser ultimately fails to parse its input — that is, when *every* possible path through the parser's assigned network leads to a dead end. Failure indicates that the parser could not process its input and, therefore, that the parser's input must be malformed (uninterpretable) in some way. The explanation of *how* the input is malformed lies in the set of dead ends that the parser reached before it failed. Some of these dead ends are reached in the parser's ordinary course of network exploration, as just described, and are not useful in identifying the practical nature of the input problem. The remaining dead ends, however, precisely identify how the input was incorrect.

VALET's parser distinguishes these "informative" dead ends from the noninformative ones by examining the parser states associated with these points. The informative dead ends are those that are associated with the "best" parser states. Parser states are ranked according to the amounts of unprocessed input that they contain: A state A is better than a state B if the list of remaining input in A is shorter than the list of remaining input in B. When two states contain equal amounts of unprocessed input, those states are compared according to "worth." The worth of a parser state is a heuristic measure of the difficulties of the actions performed by the parser in order to produce that state. As previously shown in Figure 3.5, some action definitions specify a numeric `:worth`, and by default the `:worth` of an action is 1. The worth of a parser state, then, is the sum of the worths of the actions that were successfully performed in order to produce that state. In general, the more actions required in order to produce a state, the more valuable the resultant state is. By ranking parser states by remaining input and worth, VALET's ATN parser identifies the dead ends that correspond to the most successful attempts to interpret its original input. These "best" dead ends describe the furthest limits to which the parser's input could be processed, and therefore, when the parser fails it is these points that best describe how the parser's input was malformed.

During its normal operation VALET's ATN parser remembers the set of best dead ends that it has yet encountered. The parser also remembers the state it had when it came to (and subsequently retreated from) each of those points. These states are needed in order to evaluate new dead ends that the parser may encounter, but more importantly, the state associated with a dead end must be saved so that the actions that blocked the parser's progress from that point can explain why they failed.

If the parser ultimately fails, then it can explain the reasons for its failure by consulting its saved list of best dead ends. Each dead end is a point at which one or more actions failed, causing the parser to backtrack. These failed actions are responsible for the parser's ultimate defeat. Since these failed actions thwarted the parser's most successful parsing attempts, the parser consults each of the actions for an explanation. The parser demands an explanation from an action by invoking the `:fail` part of the action's definition. (Refer to Figure 3.5. An action that has no `:fail` part cannot explain itself and is therefore never asked to do so.) When the parser invokes the `:fail` part of an action, the state of the parser is what it was when the `:body` or `:redo` part of the action failed. The `:fail` part of an action examines this state and returns a value that indicates why the action failed. In summary, in order to explain a parsing failure, VALET's ATN parser invokes the `:fail` parts of all the actions that caused the parser's best prospects to fail. The explanations provided by these individual actions combine to explain why the parser as a whole ultimately failed to interpret its input.

3.4.2.4 Summary of the ATN Parser

VALET relies on its ATN parser to interpret its users' input shell commands. When the parser succeeds, that indicates that a user's command has been completely interpreted — and possibly corrected — and is ready for final processing. VALET can send the command to the appropriate shell process, or if the command was corrected, VALET can present the revised command to its user for confirmation as described in Section 3.1.3. When VALET's parser fails, on the other hand, that means that the user's command could be neither interpreted nor corrected. In this case, through its failure explanation facility, the parser can accurately describe why it could not understand the input command. VALET can present the parser's explanation to the user and allow the user to edit the unaccepted command.

In total, VALET's ATN parser is the engine that drives the intelligent processing of C shell commands. VALET's knowledge of shell commands is embodied by the transition networks defined in VALET's Common Lisp component. Some of VALET's networks describe how shell commands should be processed in general, and other transition networks describe the syntax and semantics of specific shell commands and their arguments. The actions invoked within VALET's networks consult the system's various knowledge bases — the shell command lexicon and the file system model, for example — and invoke heuristics in order to correct specific kinds of erroneous inputs. These parser actions, outlined in Sections 3.4.4 and 3.4.5 below, link the parser's engine to VALET's contextual knowledge and thereby provide "intelligence" to the UNIX C shell interface.

3.4.3 Lexicons

In order for VALET's parser to interpret a shell command, parser actions must associate meanings with the individual words that constitute the command. VALET's knowledge bases are therefore implemented as *lexicons* — dictionaries — that map words to their definitions. VALET uses many separate lexicons, each for a particular category of terms. One lexicon contains the names of shell commands, another contains the names of files, and a third contains the login names of users. Furthermore, every shell command known to VALET is associated with its own lexicon containing the command line options (i.e., flags such as "`-c`") for that command. Lexicons are fundamental to VALET, so VALET's lexicons are represented by compact data structures that allow for speedy searches.

Because lexicons organize most of VALET's knowledge, the most important requirement for VALET's lexicons is that they allow for rapid access and searches. VALET's parser actions must be able to locate the definitions of words quickly in order for the shell interface as a whole to be responsive to its users' commands. Parser actions must also be able to locate terms within a lexicon that are similar to a given input, in case that input is erroneous and must be corrected by VALET. Another important requirement for the lexicons is that they be flexible in size. Some of VALET's lexicons hold only a few entries but others contain hundreds or thousands of items. A final criterion for VALET's lexicons is that they be compact. Because VALET uses many different lexicons of various sizes, the Lisp representation of each lexicon should be as small as possible.

For these reasons, each of VALET's lexicons is represented by a tree-like data structure. Each node in the tree is a `lex-node` structure and every link between nodes is associated with a single character. Given this structure, the function `lexicon-lookup` takes a word W and a lexicon L and then locates the definitions associated with W in L. It does this by following the links associated with the individual characters of W. Starting at the root node of the lexicon's tree, `lexicon-lookup` follows the link labeled with the first character of W. From the destination node of that link, the function then follows the link associated with the second character in W. This process of following links repeats until `lexicon-lookup` has processed all of the characters in W or until `lexicon-lookup` discovers that a required link is missing — in which case, the word W must not have any definitions in the lexicon being searched. If the `lexicon-lookup` function successfully processes all of the characters of W, then the definitions of W, if any, are stored in the `lex-node` structure that was finally reached.

The function `lexicon-insert` adds a word and its definition to a lexicon, adding any required new nodes and links to the lexicon. Because words with a common prefix share an initial sequence of nodes and links within a lexicon, the total number of nodes required to represent a lexicon is minimized and the lexicon structure is space efficient. In addition, the representation of each lexicon is further optimized by "compressing" nodes that are not shared by several words. Every word within a lexicon may be divided into a prefix that is shared by at least one other word in the lexicon and a suffix that would be represented by nodes and links unique to that word. VALET's lexicons save data space by eliminating the nodes and links that would represent the unshared suffix of each word. For each word these suffix nodes are replaced by a single `cons` cell: The `car` is a string containing the unshared word suffix and the `cdr` contains the definitions of the word. Of course, these optimized suffixes must be handled specially in the functions that modify and search VALET's lexicons.

The tree representation of lexicons allows `lexicon-lookup` to operate quickly and either locate the definitions of a given word or determine that a word has no definition within a lexicon. Similarly, because lexicon trees are organized around individual characters, it is possible for other functions to implement simple and quick spelling correction by searching for words that are similar to a given (misspelled) word. The function `lexicon-spell`, for example, searches a lexicon for defined words that are simple orthographic transformations of a word W. In particular, `lexicon-spell` suggests corrections for W based on knowledge of the most common typographic errors: the insertion, deletion, or replacement of a single character, as described in Section 2.1.3. The `lexicon-spell` function also attempts to correct capitalization errors and determines if W is a prefix (i.e., an abbreviation) of exactly one word in the current lexicon. These correction heuristics are extremely fast due to the organization of the lexicon data structures and can correct the most common kinds of input errors. However, if `lexicon-spell` is unable to suggest reasonable corrections for an erroneous term then a second function, `lexicon-guess`, can search the current lexicon for the words that are most similar to a word W according to a numerical similarity measure. This process is much slower than `lexicon-spell`, but it is rarely needed because `lexicon-spell` is almost always successful at locating accurate corrections.

The functions `lexicon-spell` and `lexicon-guess` use lexical context alone in order to correct input terms. Although this context is clearly very important it is not always sufficient (as explained in Section 2.1.4), so VALET combines lexical context with other kinds of knowledge in order to provide an intelligent shell interface. For example, in order to correct a mistyped command name VALET uses not only lexical context but also its knowledge of various commands' arguments. VALET also remembers which commands are most frequently invoked by its users and can refer to this information in order to correct malformed commands. All of VALET's information about specific shell commands resides in a single shell command knowledge base.

3.4.4 The Shell Command Knowledge Base

In a typical UNIX system there are hundreds of programs and therefore, hundreds of different C shell command names. As previously described in Section 1.3.1, most of the C shell's input language is determined by the set of external programs that are known to the shell. The first word of a shell command typically names the program to be invoked. Although this name may be an absolute or relative pathname to a program file, in general it is simply the name of a program in the shell's *search path*, a list of directories in which the shell implicitly searches for program files. The most frequently used programs are usually located along the shell's search path, so most shell commands refer to programs in this path. A typical search path, however, provides access not only to the most frequently invoked programs but also to hundreds of other programs, thereby defining hundreds of unique shell command names. This large set of commands is further augmented by various built-in shell commands and user-defined command aliases.

The language of the C shell is therefore complex because it includes several hundred different command names, most corresponding to programs along the shell's search path. In order to provide an intelligent interface to the shell VALET must know about these commands. At the very least VALET must be aware of the set of available command names, but in order to accurately correct input shell commands VALET must also have detailed knowledge of the most commonly used commands. This knowledge is stored in VALET's shell command knowledge base.

VALET's shell command knowledge base is implemented as a lexicon that associates command names with `command` data structures; the slots within each `command` structure are shown in Figure 3.6. Every shell command known to VALET is represented by its own `command` structure. Obviously, the `name` slot within a `command` contains the name of the represented command and the `from` slot indicates the source of that command.

```
(defstruct (command ...)
  (name            nil  :type simple-string        :read-only t)
  (from            nil                             :read-only t)
  (net             nil  :type (or network symbol)  :read-only t)
  (opts            nil  :type lexicon              :read-only t)
  (last-reference  0    :type fixnum                          )
  (heat            0    :type fixnum                          )
  )
```

Figure 3.6. Definition of the `command` Structure

The value of `from` is either the symbol `:built-in` (for the shell's built-in commands) or the absolute pathname of the directory that contains the program that implements the command.

The `net` slot of a `command` structure contains a transition network (or the name of a transition network) that describes the command line arguments for the represented command. Valet's ATN parser attempts to traverse this network whenever a user's input refers to the corresponding command. For example, suppose that Valet is processing the input "`cd src`". The action that interprets command names accepts the "`cd`" token, consults Valet's shell command lexicon, locates the `command` structure that represents the `cd` command, and stores the network for parsing `cd` command arguments in the `$$simple-command-net` parser variable. Once that is done the parser attempts to traverse the `cd` argument network in order to interpret the arguments ("`src`") in the current `cd` command. The `simple-command` network shown in Figure 3.4 illustrates this process.

The `opts` slot of a `command` structure contains a lexicon that describes the command line options for the represented command. Options are special arguments and are usually introduced by a "`-`" character. An option may change the behavior of a command — for instance, the "`-l`" option changes the output of the `ls` program — or an option may designate the purpose of subsequent command arguments. The lexicon in the `opts` slot of a `command` structure associates option names with data that describe the options for the represented command. The data for an option indicate whether the option may be concatenated with other options or data (e.g., whether the options "`-l`" and "`-d`" may be combined into "`-ld`"). In addition, each option is associated with a transition network that describes the command line arguments, if any, that must follow the option itself.

The final two slots in a `command` structure, `last-reference` and `heat`, indicate how recently and frequently the represented command is invoked by a Valet user. Each time VALET accepts and parses a well-formed shell command from its user, the value of the `last-reference` slot for the just-referenced command is set to the current time. In this context, the "current time" is proportional to the number of inputs that VALET has ever received from its current user, spanning all of that user's sessions with VALET. In addition to updating the `last-reference` time VALET also increases the `heat` value of the just-invoked command. "Heat" is a heuristic measure of how recently and frequently a command has been used. The heat of a command increases with each reference and diminishes slowly over time; therefore, commands that are referenced often are hotter than those that are referenced infrequently. The value of the `heat` slot in a `command` structure is the temperature of the represented command at the `last-reference` time. At later times, the function `command-current-heat` determines from this information the current temperature of the command. VALET maintains heat measures in order to establish context — to indicate which commands are most used — and thereby determine the best corrections for erroneous command names that its users may enter in the future.

The macro `define-command` describes a shell command to VALET by creating an appropriate `command` structure and adding it to VALET's command lexicon. Figure 3.7 contains VALET's definitions of three common UNIX commands: `cd`, `cp`, and `rm`. VALET has similar definitions for approximately 50 additional shell commands. This means that although VALET has detailed knowledge of the most frequently invoked shell commands, that knowledge describes only a small fraction of the hundreds of commands that are actually available to users of the C shell. It would be intolerable for VALET to limit its users to the set of commands for which VALET has detailed knowledge, however, so VALET accommodates "undescribed" programs — programs for which VALET has no explicit knowledge — by creating generic `command` structures for them. For each program that is unknown to VALET in the shell's search path, VALET creates a generic `command` structure to represent that command.[9] These generic structures are stored in VALET's shell command knowledge base along with the more detailed `command` structures created

[9]This is subject to certain restrictions. First, because VALET cannot directly determine the shell's search path, it assumes that the shell's path is equal to the path stored in the UNIX environment variable `PATH`. Second, VALET scans the search path only once for each of its users. It typically takes one or two minutes for VALET to examine a path and create the necessary `command` data structures for the programs thus found, so in order to avoid this long delay VALET scans a user's search path only during that user's initial session with the interface. VALET remembers the results of the scan for future sessions as described in Section 3.4.6. Finally, because each user's search path is scanned just once, VALET ignores relative file names (e.g., "`.`") in its users' paths.

```
;;; Definition of the "cd" (change directory) built-in shell command.  This
;;; command has special sematics that the interface must understand.

(define-command "cd"
  :from :built-in
  :net  (seq (set-semantics (change-directory $destination))
             (or (file-name (:type :directory) (:mode :executable)
                            (:into $destination))
                 (seq (no-more-command-arguments T)
                      (set-var $destination (get-file-name "$HOME" T))
                      ))
             )
  )

;;; Definition of the "cp" (copy files) command.  All the options are
;;; concatable, require no arguments of their own, and must appear before the
;;; file name arguments.

(define-command "cp"
  :from "/bin"
  :net  (seq (opt* (parse command-option))
             (or (seq (file-name (:type :regular))
                      (file-name (:type :not-directory :imaginary)))
                 (seq (one+ (file-name (:type :regular)))
                      (file-name (:type :directory) (:mode :writable)))
                 ))
  :opts (("-i" :concatable T :net no-args)
         ("-p" :concatable T :net no-args)
         ("-r" :concatable T :net no-args))
  )

;;; Definition of the "rm" (remove files) command.  Notice how the 'set-var'
;;; and 'equal-var' actions interact when a directory is to be removed.

(define-command "rm"
  :from "/bin"
  :net  (seq (set-var $rm-directories nil)
             (opt* (parse command-option))
             (one+
              (or (file-name (:type :not-directory) (:resolve-symlink nil))
                  (seq (file-name (:type :directory) (:resolve-symlink nil))
                       (equal-var $rm-directories T
                                  "use the -R option to remove a directory"))
                  )))
  :opts (("-d" :concatable T :net (set-var $rm-directories T))
         ("-f" :concatable T :net no-args)
         ("-i" :concatable T :net no-args)
         ("-R" :concatable T :net (set-var $rm-directories T))
         ("-r" :concatable T :net (set-var $rm-directories T)))
  )
```

Figure 3.7. Definitions of the `cd`, `cp`, and `rm` Commands

by `define-command`. Because VALET creates generic structures for the undescribed programs in the shell's search path, VALET's users can invoke those programs as they would in a normal command shell. However, since VALET knows nothing about the command line arguments required by those undescribed programs, VALET is unable to correct errors that users make in the arguments given to those programs.

VALET could treat user-defined command aliases in the same way that it treats undescribed commands. However, whereas it is relatively easy for VALET to determine the set of programs in the C shell's search path, it is much more difficult for VALET to determine the set of aliases that its users define. VALET's "intelligent" Common Lisp component is separate from the managed C shell processes, so VALET does not have direct access to the shells' internal states, including the sets of defined aliases. VALET could conceivably determine its users' aliases by parsing its users' "`.cshrc`" shell configuration files, but that parsing task would be complicated. It would require VALET to understand all of the programming constructs of the C shell language (e.g., conditional statements), and that level of knowledge is much more than VALET needs in order to interpret users' interactive commands. Even if VALET simply extracted alias definitions without truly parsing its users' "`.cshrc`" files, problems would still arise: VALET would still be required to parse alias definitions (not always a trivial task) in order to associate semantics with certain aliases. For example, VALET would have to recognize and interpret aliases that change the shell's current directory. Therefore, due to all of the difficulties involved, VALET does not create `command` structures for aliases and this means that the command parser cannot recognize invocations of aliases.[10] This shortcoming is a nuisance that results from VALET's experimental architecture. Fortunately it does not seriously hurt VALET's ability to interpret and correct most common shell commands.

Even without command aliases, VALET's shell command knowledge base contains hundreds of entries. The knowledge base stores information about all the programs that may be invoked through the shell's command search path, and in addition, the knowledge base details the command line arguments for many of the most frequently invoked shell commands. The ATN parser action that interprets command names consults VALET's command knowledge base and uses context in order to correct misentered command

[10] Although VALET cannot determine its users' aliases for itself, it was hoped that VALET would allow its users to manually describe their aliases to the interface. Unfortunately, that ability was not implemented before the user testing experiment described in Chapter 4 was performed. In retrospect, given the importance that users placed on their personal aliases, VALET should have handled aliases more gracefully.

names. This parser action makes use of lexical context by invoking the functions described in Section 3.4.3 to find command names that are similar to an erroneous command name. These possible corrections are then evaluated in light of other contextual data stored in VALET's command knowledge base: the frequency at which various shell commands are referenced and the arguments that each command requires. Once the parser has digested the name of an input command, subsequent ATN parser actions interpret command line arguments by referring both to data stored in VALET's shell command knowledge base (e.g., the appropriate command option lexicon) and also to data stored in VALET's other knowledge bases. Most notably, since so many command arguments refer to files, VALET's command argument parsing actions make heavy use of VALET's internal file system model.

3.4.5 The File System Knowledge Base

Knowledge of its UNIX host's file system is critical to VALET. The file system is one of the most fundamental and user-visible components of a UNIX computer system because it provides the principal metaphors through which the operation of the entire system is understood. Most of the C shell's command language is determined by the file system; not only are the shell's commands determined by the programs located along the shell's search path, but more importantly, many command line arguments name files: existing files to be read, new files to be written, files to be created, consulted, moved, modified, or destroyed. Therefore, in order for VALET to understand its users' shell commands it must have intimate knowledge of its host computer's file system. This knowledge is contained in a sophisticated file system knowledge base.

VALET's file system knowledge base is complex. The code that implements the knowledge base includes approximately 1,500 lines of Common Lisp code and 275 lines of C code — and these figures refer only to the code that creates and maintains the knowledge base itself. The parser actions and functions that actually consult the knowledge base amount to another several hundred lines of Lisp code. The complexity of VALET's file system knowledge base is due to the nature of UNIX file systems:

- A typical UNIX file system contains hundreds of directories and thousands of files, all organized in a single hierarchy. VALET must model these objects and their organization.

- The organization of a file system changes almost continuously. New directories and files are created and existing directories and files are modified, renamed, relocated, or destroyed. This constant process of change means that VALET must frequently update its file system knowledge base. Unfortunately, VALET is not informed when its host's file system changes — and VALET certainly cannot control such changes — so it is difficult for VALET to update its file system model appropriately. The result is that VALET's model often contains outdated information.

- Examination of a UNIX file system is very time consuming, so VALET must be judicious when it creates and updates its file system model. It is not feasible for VALET to represent its host's entire file system; the storage size of the model would be too great and the time required to create the model would render the interface unusable. (Further, due to UNIX file access restrictions, it is generally not possible for VALET to construct a complete file system model in any case.) For these reasons VALET's file system knowledge base is incomplete. The model represents only the portions of the file system that VALET needs in order to establish context and interpret its users' input shell commands. Of course, even this partial representation needs to be updated regularly in order for VALET to track changes that occur in the file system. Because examination of the file system is so expensive, however, VALET is careful to update information only as it is needed. Furthermore, VALET minimizes its examination of the actual file system by remembering which parts of its knowledge base are current and which are not. In effect, VALET's file system knowledge base acts as a cache of information about the actual file system.

- Finally, the Common Lisp language [31] does not contain functions that would allow VALET's Common Lisp component to examine its host's file system in sufficient detail. Therefore, VALET communicates with an external process in order to build its file system model.[11] The need to communicate with an external UNIX process complicates the implementation of the knowledge base and makes examination of the actual file system even slower, thereby increasing the need for VALET to cache information and minimize its inspection of its host's file system.

---

[11] The Common Lisp language does not specify functions for communicating with external processes, either, but most Common Lisp implementations for UNIX provide such functions.

In short, although VALET needs a great deal of information about its host's file system, this knowledge is expensive to acquire and must be continually refreshed. VALET's file system knowledge base was therefore implemented very carefully in order to allow the interface to interpret and correct file names with acceptable speed.

3.4.5.1 Representation of the File System

VALET's file system knowledge base is implemented by three different organizational data structures. The first of these is a tree of linked Common Lisp structures that mirrors the hierarchical organization of the actual file system. The definitions of the structures in this tree are shown in Figure 3.8. For every unique file known to VALET there is a corresponding `file` structure that represents and describes that file. In UNIX terms, a `file` structure contains information gathered from a file's *inode*: the file's type and size, the identity of the file's owner, the access permissions associated with the file, and so on. The `scan-time` slot in a `file` structure indicates the time at which the inode information was gathered; the value of `scan-time` is proportional to the number of shell commands that VALET has ever received from its current user. It is important to realize that a `file` structure does *not* specify the name of the file that it represents. In a UNIX file system, the association of names to files is controlled by directory entries as described below.

Most files in VALET's file system model are represented by `file` structures, but symbolic links and directories are represented by instances of special subtypes of the `file` structure. A symbolic link is a file that contains the name of another file; VALET represents symbolic links by special `symlink` structures in its file system model. A directory is a second kind of special UNIX file. Every directory contains a list of names, and each of those names is associated with a particular file in the file system. In VALET's file system model, each directory is represented by a `dir` structure. In addition to the normal `file` slots, each `dir` structure contains three additional slots:

1. The `children` slot contains a list that describes the entries within the represented directory. This is a list of `dirent` structures as described below.

2. The `children-current-p` slot is a flag that indicates whether or not the entries in the `children` list are known to be correct as of the time stored in the `dir` structure's `scan-time` slot. VALET's file system knowledge base often updates the information about a directory (i.e., the values in the `dir`'s `file` slots) without immediately

```
;;; A ‘file’ contains stat(2) information about an actual file.  This data is
;;; timestamped by Valet when it is acquired.

(defstruct (file ...)
  (type         nil  :type symbol   :read-only t) ;; :regular, :directory, ...
  (device       0    :type fixnum   :read-only t) ;; File device #.
  (inode        0    :type integer  :read-only t) ;; File inode #.
  (mode         0    :type fixnum               ) ;; UNIX permission bits.
  (links        0    :type fixnum               ) ;; # of links to this file.
  (owner        0    :type fixnum               ) ;; UID of owner.
  (group        0    :type fixnum               ) ;; GID of group.
  (size         0    :type integer              ) ;; File size in bytes.
  (access-time  0    :type integer              ) ;; Time of last access.
  (modify-time  0    :type integer              ) ;;   ...modification.
  (change-time  0    :type integer              ) ;;   ...change in inode.

  (scan-time    0    :type fixnum               ) ;; Time of Valet’s scan.
  )

;;; A ‘symlink’ is a special ‘file’ that represents a symbolic link.

(defstruct (symlink (:include file) ...)
  (contents     nil  :type (or simple-string null)) ;; Contents of this link.
  )

;;; A ‘dir’ is a special ‘file’ that represents a directory.  A directory
;;; contains entries represented by ‘dirent’ structures.

(defstruct (dir (:include file) ...)
  (children           nil  :type list  ) ;; The entries in this directory.
  (children-current-p nil  :type symbol) ;; Is the ‘children’ list current?
  (parent             nil              ) ;; Link to entry in our parent dir.
  )

;;; A ‘dirent’ represents an entry in a directory.  An entry associates a name
;;; (a string) with a file in the file system, or with an "imaginary" file.

(defstruct (dirent ...)
  (name            ""   :type simple-string               )
  (file            nil  :type (or file null)              )
  (parent          nil  :type dir             :read-only t)
  (last-reference  0    :type fixnum                      ) ;; Valet timestamp
  (heat            0    :type fixnum                      ) ;;   and heat val.
  )
```

Figure 3.8. Definitions of the File System Model Structures

updating the information about the contents of that directory. In these cases VALET may need to remind itself to later reexamine the list of directory entries.

3. The `parent` slot of a `dir` structure is a link to the "parent" of the represented directory. More specifically, the `parent` slot of a `dir` D is a link to the `dirent` structure that refers to D in the `dir` structure above D in VALET's modeled file system hierarchy. This `parent` link makes it possible for VALET to immediately determine the name associated with a given directory.

Every directory in a UNIX file system contains entries that give names to files. In VALET's model of the file system, the contents of a directory are represented by a list of `dirent` structures within that directory's `dir` structure. Each `dirent` contains a name, unique to the list in which the `dirent` is contained, and a reference to the `file` structure to which the entry refers. Each `dirent` also contains three additional fields: a link to the `dir` in which the `dirent` is contained and two timestamps that VALET associates with the entry. These last two fields, `last-reference` and `heat`, are similar to the timestamp slots contained in VALET's `command` structures, previously explained in Section 3.4.4. Every time an input shell command refers to a directory entry, the `last-reference` slot of the corresponding `dirent` is updated and the `heat` of the `dirent` is increased. This provides important context that VALET uses in order to correct file names in the future. It is important to understand that VALET associates user reference information with directory entries — essentially, with file *names* — and not with the actual files referenced by those entries. This distinction is important because it is file names, not file inodes, that are meaningful to users of the C shell. When the entries within a directory change, VALET takes great care to associate appropriate reference information with each of the corresponding updated `dirent` structures in its file system model.

Because users' references to file names (directory entries) are so important for establishing interaction context, VALET maintains reference information not only for names that refer to existent files but also for many names that refer to nonexistent files. It is not at all uncommon for a shell command to contain file names that are not valid before the command is executed; for instance, when a file is renamed with the `mv` command, it is often the case that the new name for the file does not refer to any file at the time the `mv` command is invoked. The new name for the file can be understood as a reference to a directory entry that *will* exist in the near future, after the completion of the `mv` command.

For situations such as this, VALET's file system model can create "imaginary" directory entries.

An imaginary entry is essentially a file name that does not correspond to any file in the host's actual file system. In VALET's file system model, an imaginary directory entry is represented by a `dirent` structure that has a name but no associated `file` structure; the `file` slot of an imaginary `dirent` is `nil`. By creating imaginary entries, VALET's file system knowledge base can keep track of users' references to file names even when those file names are invalid at the times at which they are entered. This reference information is saved in expectation that the file names will become "actual" in the near future. When the directory entry represented by an imaginary `dirent` actually appears in the real file system, the `dirent` is updated to refer to the appropriate `file`, `dir`, or `symlink` structure. VALET's information about references to that previously imaginary entry is preserved.

VALET's file system knowledge base assembles instances of the data types described above — `file`, `symlink`, `dir`, and `dirent` structures — into a single tree-like hierarchy that models a portion of the actual file system of VALET's host.[12] At the top of this tree is a single `dirent`, the `*root-dirent*`, which corresponds to the UNIX file name "`/`". Below the `*root-dirent*` are the portions of the file system that VALET needs in order to establish context and interpret its users' input shell commands. This tree describes the "spatial" organization of the actual file system, and VALET's file system knowledge base needs two additional organizations in order to serve the intelligent interface. The first of these is a hash table that allows the knowledge base to locate the `file`, `dir`, or `symlink` structure that corresponds to a particular inode and device number pair; this table is internal to the knowledge base itself. The second organizational structure is a lexicon (the structure described in Section 3.4.3) that maps file names to the `dirent` structures that contain those names. Through this lexicon the knowledge base can quickly locate all of the files with a given name or locate all of the files that have names similar to a misspelled name. By incorporating both a hierarchical model and a lexicon, the file system knowledge base offers both "spatial" and lexical context that VALET uses in order to correct erroneous file names. The reference information within the model's `dirent` structures provides additional context to the intelligent interface. All of the information

[12]The hierarchy is not entirely acyclic, however, because the UNIX file system itself is not entirely acyclic. In particular, the "`.`" and "`..`" entries within every UNIX directory introduce cycles into the file system and into VALET's model.

in the file system knowledge base is dynamic and must be continually updated as new data are received from VALET's `flunkie` process.

3.4.5.2 Examination of the File System

VALET's file system knowledge base relies on a separate UNIX process, the file system `flunkie`, to provide information about the host's actual file system to VALET's Common Lisp process. When the file system knowledge base determines that it must refresh part of its file system model, it sends a request to the `flunkie`, asking that the `flunkie` reexamine the appropriate parts of the file system. The `flunkie` in turn responds with data about the current states of the examined objects, and VALET incorporates this new information into its file system model. (In addition to information about the file system, the `flunkie` also relays information about the users of the UNIX system — their home directories and their user and group identification numbers — to VALET's Common Lisp component.)

Examination of a UNIX file system is inherently time consuming and the need for VALET's Common Lisp process to communicate with a separate `flunkie` process makes examination of the file system even slower. Unrestrained perusal of the actual file system would immediately bring the entire VALET system to a crawl and make the intelligent shell interface intolerably slow. Therefore, VALET's file system knowledge base takes great care to remember which parts of its internal model are up to date and which are not. As described in Section 3.4.5.1, every file in the model is associated with a `scan-time` and every directory in the model contains a `children-current-p` flag that indicates whether the entries within the directory are known to be current. By consulting these data and by updating its model only as necessary, VALET's file system knowledge base minimizes the communication between itself and the `flunkie`, thereby allowing the knowledge base to operate as quickly as possible. This in turn allows VALET's `file-name` ATN parser action to interpret and correct file names speedily, even when it must consult large amounts of context in order to do so.

3.4.5.3 Parsing and Correcting File Names

A single ATN parser action called `file-name` is responsible for interpreting and correcting file names that appear in input shell commands. Not surprisingly, because file names are used in so many different circumstances and because there is so much context to consider when a file name is to be corrected, the `file-name` parser action is the single most complicated parser action defined in VALET. The `file-name` action accepts many different arguments as summarized in Table 3.4. The command definitions in Figure 3.7 illustrate how the `file-name` action is actually used in command networks.

When the `file-name` action is invoked, it takes the next available token from the ATN parser's input and tries to interpret that token as a file name. The action consults the file system knowledge base and attempts to locate the `dirent` structure that corresponds to the current input token; when the search is successful, the `file-name` action adds the file name token and the named `dirent` to the output of the parser. The arguments listed in Table 3.4 control how the `file-name` action operates. Most importantly, the `:type` argument lists the kinds of files to which the current file name may refer. The set of possible types includes the standard UNIX file types (`:regular`, `:directory`, and so on), the aggregate types `:not-directory` and `:any-existent-type`, and the special type `:imaginary`. As described in Section 3.4.5.1, an imaginary directory entry is one that does not currently exist in the actual file system; the type `:imaginary`, then, tells the `file-name` action that the current file name may refer to an imaginary `dirent`. (The `:imaginary` keyword gives the `file-name` action license to create a new imaginary `dirent` in VALET's file system model if necessary. Note, however, that the directory in which the imaginary `dirent` is to be created must itself exist.) The default value of the `:type` argument is `:any-existent-type`.

The `:resolve-symlink` argument determines what the `file-name` action will do if the current file name refers to a symbolic link: Should `file-name` return the directory entry for the file named by the link (i.e., "resolve" the link) or should `file-name` return the directory entry for the link itself? The `:mode` argument specifies the set of properties that the named file must have: whether the file must be `:readable`, `:writable`, or

Table 3.4. Summary of the `file-name` Parser Action Arguments

Argument	**Meaning**
`(:type ...)`	Acceptable file types (e.g., `:regular`).
`(:resolve-symlink` *t-or-nil*`)`	If name refers to a symbolic link, resolve it?
`(:mode ...)`	Required file modes (e.g., `:readable`).
`(:prefer-extension ...)`	Preferred file name extensions (e.g., "`.c`").
`(:provide-extension ...)`	Extensions that may be provided.
`(:into ...)`	Variables in which to store the `dirent`.
`(:accumulate-into ...)`	Variables in which to store the `dirent`.

`:executable` (in any combination) by VALET's user.[13] The `:prefer-extension` argument contains a list of symbols that describe file name extensions (i.e., final name segments) and the current file name is expected to have one of these extensions. None of the listed extensions are required to be present in the file name, but the `:prefer-extension` list provides useful contextual data in case the file name needs to be corrected. The `:provide-extension` list is the set of extensions that the `file-name` action is allowed to append to the current file name in order to locate the referenced directory entry. These do not apply when the current file name already has an extension, however. Finally, the `:into` and `:accumulate-into` arguments name parser variables into which the `file-name` action should store the `dirent` that corresponds to the current input token. The variables in the `:into` list are set to contain the located `dirent`. The variables in the `:accumulate-into` list are assumed to already be bound to list values, and the `dirent` located by the `file-name` action is inserted into each of those lists.

Given values for some or all of the above arguments, the `file-name` action attempts to interpret the parser's current input token as a file name. If the token correctly names a directory entry then the `file-name` action succeeds or fails depending on the type and mode of the referenced file. (Unless explicitly told to do so, however, the `file-name` action will not allow the input file name to refer to an imaginary directory entry.) If the input token does not correspond to a directory entry, however, the `file-name` action attempts to correct the original file name so that it refers to an appropriate file. The correction heuristics within the `file-name` action are driven by context.

The first step in correcting a file name is to interpret the nonterminal components of the name. Nonterminal components are directory components; for instance, in the file name "`local/src/main.c`" the segments `local` and `src` are nonterminal components that are expected to name directories in the file system hierarchy, and the order of these components reflects the expected file system structure. When the name given to the `file-name` action must be corrected, the `file-name` action examines the nonterminal components of that file name and produces a set of directories around which it will later search for the file that best corresponds to the final segment of the name. This set of *startpoint* directories is constructed as follows:

1. If the original file name is a relative file name, the *startpoint* set is initialized to

[13] Imaginary files are treated as `:writable` but neither `:readable` nor `:executable`.

contain the shell process' current directory. Otherwise, the file name is an absolute file name and the *startpoint* set is initialized to contain the root directory.

2. The nonterminal components of the file name are processed in order, and the set of *startpoint* directories is updated with each step through the list of components. For each component:

   (a) The "neighborhood" of each directory in the *startpoint* set is searched for directories that have names that are identical to or simple variations of the current file name component. "Simple" variations are those that can be corrected by `lexicon-spell` as described in Section 3.4.3. The neighborhood of a directory includes (1) the directory itself, (2) the parent of the directory (i.e., the directory that is above the current directory in the file system hierarchy), (3) the children of the directory (i.e., the directories that are immediately below the current directory in the file system hierarchy), and (4) the siblings of the directory (i.e., the other directories that are below the parent of the current directory). Roughly speaking, by searching the neighborhood of each directory the `file-name` action allows for extra, omitted, and incorrect components within file names. If the search locates any appropriately named directories, the set of *startpoint* directories is set to contain the newly found directories and the control returns to the beginning of step 2 in order to consider the next file name component.

   (b) If, however, the previous search failed to locate any directories at all, then the neighborhoods of the *startpoint* directories are searched again, this time for directories that have names similar to the current file name component as determined by the function `lexicon-guess`. If this search locates any directories, then these are the new *startpoint* directories and control returns to the beginning of step 2.

   (c) If the previous searches failed to locate any directories, then VALET's entire file system model is searched for recently referenced directories with names that are identical to or simple variations of the current file name component. Because all directory entries are stored in a lexicon, it is easy and quick to search the entire model by invoking `lexicon-spell`. Again, if any directories are located, they become the new *startpoint* set and control returns to the start of step 2.

(d) Finally, if all of the previous searches failed, then the entire file system model is searched for recently referenced directories with names that are similar to the current file name component as determined by `lexicon-guess`. The set of located directories — even if that set is empty — becomes the new *startpoint* set and control returns to the top of step 2.

When the `file-name` correction heuristic searches the neighborhood of a directory, it must consult many segments of VALET's file system knowledge base. Each examination could potentially cause the knowledge base to communicate with the `flunkie` process; left unchecked, the need to update large parts of the file system model would make the correction of file names intolerably slow. Therefore, portions of the neighborhood searches described above are coded to examine only data already in VALET's model. (This constraint ultimately proved to be too severe as explained in Section 4.2.4.)

After all the nonterminal components of the current file name have been processed as described above, the final *startpoint* set is the set of directories around which the `file-name` action must search in order to locate the referent of the final file name component. This process is similar to the search just described. The correction heuristic searches the neighborhood of each *startpoint* directory for entries that have names identical or similar to the final file name component, first directed by the function `lexicon-spell` and then by `lexicon-guess`. As allowed by the `:provide-extension` argument, the `file-name` action may append various extensions to the final file name component in order to find a matching directory entry. If the neighborhood searches fail then the `file-name` action searches VALET's entire file system model for appropriate directory entries. Ultimately the final set of located directory entries represents the likely corrections for the original, incorrect file name. From this set the `file-name` action chooses the best candidate correction as determined by context: the set of acceptable file types, the required file modes, the preferred file name extensions, the `heat` values associated with the candidate corrections, and the "distance" to each correction (i.e., the number of components in the absolute or relative file name required to reference each potential correction).

If VALET's `file-name` parses its given input file name — either verbatim or through the heuristics designed to correct erroneous file references — and successfully locates a directory entry that refers to a file of the required type and mode, the `file-name` action adds the parsed and possibly corrected file name to the ATN parser's output, along with the `dirent` structure to which the name refers. Then, if the parser successfully interprets the remainder of the user's input command, VALET updates the `dirent` structures in the parser's output with new values for their `last-reference` and `heat` slots. This new reference information provides context for future commands, both for the current VALET session and for future sessions. By preserving contextual data through many separate sessions, VALET's knowledge bases become tuned to the behaviors of its individual users and the intelligent shell interface as a whole becomes increasingly proficient at properly correcting its users' input errors.

3.4.6 Preservation of Interaction Contexts

Much of VALET's knowledge describes fixed, unchanging aspects of the UNIX operating system and the UNIX C shell. For instance, VALET knows the command line syntax required by the `ls` program and this information is applicable to all invocations of `ls` — the `ls` command does not behave differently for different people. This kind of knowledge, constant across time and across different users of the C shell, is built into VALET. On the other hand, VALET also needs knowledge that is specific to each of its users. For example, in order to correct shell commands accurately, VALET remembers the set of recently invoked commands and the set of recently referenced file names. These sets are naturally different for each user of the interface, and in addition, these sets change over time. Not only do these sets provide context in the short term, but over the course of hundreds of shell commands these sets come to describe the long-term habits of VALET's individual users. Therefore, in order to establish long-term interaction context for each of its users, VALET preserves its user-specific data across separate invocations of the interface. When a user ends a session, VALET saves the information specific to that user in a special file. Later, when that user reinvokes the interface, VALET reads the user's file and restores the user's interaction context.

VALET stores user-specific contextual data in files within its users' home directories. Each user's "`.valetrc-`*host*`.l`"[14] file contains forms that preserve the "current time," the shell command knowledge base described in Section 3.4.4, and the file system knowledge base described in Section 3.4.5.

- The saved time is proportional to the number of shell commands that the user has ever entered to VALET. This time must be preserved and restored in order for VALET

[14]The *host* portion of the file name is the name of the machine on which VALET runs.

to interpret the `last-reference` and `heat` data within the shell command and file system forms described below.

- Every command in VALET's shell command knowledge base is described in the user-specific context file. For each command VALET remembers the command's name and location (i.e., the symbol `:built-in` or name of the directory that contains the named program file) and the `last-reference` and `heat` values associated with that command. These saved data, when combined with VALET's built-in knowledge of certain commands, make it possible for VALET to restore its shell command knowledge base completely. In particular, the saved command data generally make it unnecessary for VALET to scan the shell's search path, so the VALET session can start quickly.

- Information about "hot" directory entries is also preserved in users' context files. At the end of each user session VALET searches its file system model for all the directory entries that are still "hot" according to their `last-reference` and `heat` values. For each hot entry VALET remembers the absolute file name of the entry and the `last-reference` and `heat` values for that entry. Later, when the user's saved context is restored, VALET recreates the appropriate portions of its file system model in order to restore the `last-reference` and `heat` values for all the directory entries described in the user's context file. In the process VALET also recreates the "spatial" context around each of those entries in the file system model. Overall, the effect is that VALET recreates all of the recently referenced portions of its file system model every time the intelligent interface is invoked.

All of the above contextual information is encoded in order to be compact and quickly interpretable. Every time a new VALET session begins, VALET's Common Lisp process loads its user's "`.valetrc-`*host*`.l`" file, and the forms in that file invoke special functions that recreate the current user's context. This context in turn allows VALET to refer to its user's past commands in order to accurately correct his or her future input errors.

3.5 The Process of Input Correction

Through knowledge and context, VALET's Common Lisp components — the input tokenizer, the ATN parser, and the various knowledge bases — attempt to interpret and correct commands entered to the C shell. Consequently, the extent to which VALET can successfully understand its users' inputs is limited by the knowledge contained in the interface. VALET incorporates a great deal of information such as the set of available command names, descriptions of the command line arguments required by popular commands, an understanding of file name patterns, data about the file system, data about the users of the host UNIX system, and timestamps that indicate how frequently various entities have been referenced in users' input commands. All of this knowledge is essential in order for VALET to interpret the kinds of commands that people enter most frequently. In order to interpret *all* possible shell commands intelligently, however, VALET would need access not only to the kinds of information just listed but also to many more obscure domains. Some of this data would be very difficult or impossible to acquire (e.g., the names of all users of a remote UNIX system), and even if VALET had access to these data they would be generally useful for only a relatively small number of user inputs. For these reasons VALET concentrates on the more important and more common cases. VALET's knowledge bases describe the domains that are most important to the interpretation of shell commands and VALET's heuristics allow the interface to correct the most frequently occurring errors in those commands.

In order to interpret an entire command VALET must associate meanings with each of the individual terms that constitute that command. This means that VALET must determine the domain to which each term refers (i.e., discover the kind of thing named by each term). This associative process is carried out by VALET's ATN parser and its actions. For each term, the set of domains to which that term may belong is determined by the "position" of VALET's ATN parser within its transition network at the time that term must be interpreted. For example, from a certain position the parser may have a choice between two actions: one that attempts to parse the next input term as a command option and another that attempts to parse the next input term as a file name. In general, the set of acceptable domains for a term is determined by the set of currently available actions, and that set of actions is determined by the syntax of the current command as embodied in that command's transition networks. From the available set of actions the parser discovers the domain of its current input term through trial-and-error execution of the actions; a parser action succeeds only if the current term appears to belong to the domain examined by the action.

In order to interpret an input term reasonably, a parser action must be able to determine if that term is a member of the action's domain, either through a priori knowledge (e.g., the set of options for a command) or through access to external sources of information (e.g., the file system). This allows an action to recognize terms within its domain but does not necessarily allow the action to correct malformed terms. In order to correct a term an action must be able to do more than simply verify particular names: It must also be able to search its domain for terms that are similar to a malformed term, and ideally, it should be able to use various kinds of context in order to locate the best possible corrections. In VALET's model, a fully implemented action has access to a lexicon that completely describes the action's domain, and further, it uses both general and domain-specific context in order to correct malformed terms. Ideally there would be a unique parser action for every possible domain and each of those actions would have complete knowledge of the domain to which it refers. Unfortunately, neither of these situations is possible. Due to the nature of the C shell language and the unlimited number of domains to which that language may refer there will always be some domains for which VALET has incomplete information. Therefore, some of VALET's parser actions fully implement VALET's interpretation and correction model and some do not. In short, VALET "understands" different domains to differing degrees.

Some domains — shell command names and file names, for instance — are vital to understanding practically all shell commands, so for these domains VALET has very detailed knowledge bases and sophisticated correction heuristics. VALET acquires data for these essential kinds of terms even when those data are complex or difficult to acquire, and the result is that within these domains VALET is very powerful. For example, unlike the `metric` library [13] and `tcsh` [27] and `zsh` [7] shells described in Chapter 2, VALET can correct file names that contain extra or missing or even completely wrong directory components. Furthermore, because VALET remembers context that those other systems ignore, VALET can search for corrections based on file types and according to the frequencies at which various files have been previously referenced.

The file system is critical to understanding shell commands, but many other domains are not so important or are too difficult for VALET to examine practically. For instance, in order for VALET to interpret or correct arguments given to the UNIX `make` program, VALET would need a massive amount of code devoted just to `make`. Similarly, in order to verify electronic mail addresses VALET would need to communicate with remote computer systems. Although the ability to understand `make` arguments and mail addresses would be very useful in certain circumstances, the effort that would be required in order for

VALET to understand these domains would be enormous and only useful for a relatively special set of shell commands. The value gained is far outweighed by the effort required, so for many less important kinds of inputs VALET implements only a partial or minimal interpretation model.

In addition to these limits on interpreting individual words, VALET has limits on its ability to interpret certain kinds of complete commands. Most obviously, in order to accurately interpret the arguments within an input command, VALET's shell command knowledge base must have a description of the expected command line syntax. This description is embodied by a transition network particular to the invoked command. Most of VALET's command argument networks accept only the syntax required by the corresponding commands, but it would be possible for these networks to accept more flexible kinds of inputs. It would be possible, for example, for VALET to define transition networks that could reorder command arguments — in order to move all options to the beginning of the argument list as required by some UNIX programs — or insert or delete tokens in order to make sense of a command line. These features would be occasionally useful but are almost entirely unimplemented because they do not correspond to the most frequently occurring types of input errors.[15] For similar reasons VALET understands the command line syntax required by only a small subset of the hundreds of available UNIX programs: The effort required to describe *every* command, not just the most popular ones, would be very large and would reap additional rewards only in relatively seldom cases. Rather than attempt to handle every circumstance, VALET instead concentrates on the most common kinds of shell inputs and the most useful correction heuristics.

In conclusion, although its ability to interpret and correct shell commands has certain limits, VALET nonetheless serves to demonstrate the effectiveness of user-supportive, context-based, "intelligent" user interfaces. The error correction features that VALET incorporates are those that are generally most useful to experienced users of the C shell.

[15]The `literal` parser action, which looks for and accepts a particular token, may try to insert an expected token into a command.

CHAPTER 4

EVALUATION

The true test of a human-computer interface is the effectiveness with which that interface meets the needs of its users. VALET attempts to address its users' needs by adding "intelligent" command analysis to the UNIX C shell. VALET's goal is to provide an intelligent, user-supportive interface to the C shell, an interface that uses knowledge in order to interpret and as necessary correct its users' shell commands. In order to evaluate how well VALET meets this goal, 11 people were recruited to use the VALET interface in the course of their everyday work. The commands that these people entered, along with VALET's interpretations and responses, were recorded in transcripts. These transcripts were later analyzed in order to evaluate the interface and answer questions such as the following:

- How often do people enter erroneous shell commands? Because VALET's principal feature is that it corrects errors, the frequency at which errors are made imposes a limit on VALET's overall usefulness.

- What kinds of errors do people actually make, and with what frequencies? This information determines the importance and applicability of VALET's various input correction heuristics.

- How often does VALET properly detect errors? Conversely, how often does VALET fail to detect errors (i.e., how common are "false negatives") and how often does VALET incorrectly decide that a truly valid command contains an error (i.e., how common are "false positives")? In addition to providing data for evaluation of the current system, an understanding of the causes of false positives and negatives is important for possible future improvement of the interface.

- When an actual mistake is detected, how often can VALET infer its user's intended input? How often is the interface unable to offer any correction at all to its user, and why is Valet unable to infer reasonable corrections in these cases? The answers to these questions describe the effectiveness of Valet's current correction heuristics and also provide insight for future improvements.

The data gathered during the user testing experiment answer the above questions and highlight both Valet's strengths and weaknesses. Section 4.2 below presents a detailed analysis of the data, but in brief the results suggest that Valet's style of intelligent parsing and input correction can be a very effective component of a user interface. Valet corrected slightly over half of all the erroneous shell commands that were entered during the user testing experiment, and as described in Section 4.2.4, in most but not all of these cases Valet's suggested correction was accepted verbatim by Valet's user. Most of Valet's deficiencies can be attributed to its lack of certain kinds of knowledge, correction heuristics that were sometimes too conservative, and other restrictions of the current implementation — in other words, factors that could be alleviated in future intelligent shell interfaces.

4.1 The Experiment

In order to understand Valet's performance in "real world" situations, members of the University of Utah Department of Computer Science were recruited to use Valet. The study of a new "intelligent" shell interface was announced through electronic mail and news and 25 people responded. Of those, 11 eventually participated in the testing of Valet. Most of the rejected volunteers either never used the computer systems on which Valet ran or were not available to use the interface during the predetermined testing period.

The 11 volunteer participants in the study included undergraduate students, graduate students, and faculty members from the Department of Computer Science. Each of the participants was already familiar with the C shell and with GNU Emacs, and each agreed to use Valet in the course of his or her normal work during the period of the study, approximately four weeks. The participants were told that Valet was designed to interpret shell commands and correct input errors and that the purpose of the experiment was to study Valet's abilities in natural situations. Therefore, despite Valet's abilities the testers were asked to behave normally and not to make intentional input errors. Each participant received a brief set of instructions for Valet describing how to start the interface and how to invoke the system's special features (e.g., the command history described in Section 3.3.1). The instructions also listed the kinds of inputs that VALET could not understand: aliases, references to the C shell's built-in command history (e.g., the command "!!"), references to shell variables, and other certain constructs. In addition to instructions, each VALET user also received and signed an informed consent form. This form described the purpose of the experiment and stated that all of the shell commands input to VALET would be recorded and later analyzed in order to measure the overall effectiveness of VALET's intelligent features.

The VALET system was available for a period of approximately four weeks during July and August 1993, and in that time VALET recorded information about 1,126 nonempty inputs to the shell from the group of 11 testers. At the conclusion of the testing period each user was asked to fill out a brief questionnaire in order to record his or her general impressions and comments about the interface. None of the VALET testers received any compensation for taking part in the experiment.

4.2 The Results of the Experiment

Table 4.1 summarizes VALET's handling of the 1,126 shell commands that were analyzed and recorded during the user testing experiment. Each input falls into one of three categories representing VALET's three types of response:

1. **Accepted.** An accepted input is one that VALET determined to be correct — in other words, an input that VALET parsed and sent to the underlying C shell process verbatim. As described below, a few accepted inputs were actually erroneous; these inputs contained errors that VALET failed to recognize.

Table 4.1. Distribution of Correct and Incorrect Inputs Across VALET's Responses

	All Recorded Inputs		**All Correct Inputs**		**All Incorrect Inputs**		**Unintent'ly Incorrect Inputs**	
Response	#	%	#	%	#	%	#	%
Accepted	994	88.3	983	94.0	11	13.75	7	15.2
Corrected	60	5.3	19	1.8	41	51.25	18	39.1
Rejected	72	6.4	44	4.2	28	35.00	21	45.7
Total	1126	100.0	1046	100.0	80	100.00	46	100.0

2. **Corrected.** A corrected input is one that VALET recognized as erroneous and for which VALET offered a correction to the user. For example, in one case, one of the VALET testers entered the shell command "`cd par`". VALET discovered that the directory `par` was not in the shell's current directory but that `par` was in fact one of the siblings of the current directory. VALET therefore presented a correction of the original command to the user, suggesting that "`cd ../par`" was most likely what the user intended. Obviously, whenever a user's input was corrected, that input was not sent to the underlying C shell process. VALET occasionally offered corrections for commands that were not in fact erroneous, as described below.

3. **Rejected.** A rejected input is one that VALET recognized as erroneous but for which VALET offered no correction. Instead, VALET simply presented an appropriate error message to the user. (The construction of these error messages was detailed in Section 3.4.2.3.) Obviously, rejected inputs were not relayed to the C shell process for execution. As with the set of corrected inputs, not all of the rejected inputs actually contained errors.

Of the 1,126 total recorded inputs, 994 inputs (88.3%) were accepted by VALET. In addition, 60 inputs (5.3%) were corrected and 72 inputs (6.4%) were rejected. The total count of inputs excludes "empty line" inputs, which are normally ignored by the C shell, and also excludes approximately 232 inputs for which no parsing information was recorded. Due to technical problems, information about those inputs was lost.

Overall, the number of recorded inputs was substantially less than the number that was anticipated because several of the 11 volunteers used VALET on only a few occasions. Table 4.2 summarizes the number of commands entered by each of VALET's testers. Clearly, each user contributed to a differing extent — user A by himself entered almost half of all the recorded commands. Some of the numbers in Table 4.2 are lower than they should be because certain inputs were not recorded, as just described. The missing commands include approximately 56 inputs from user A, 26 from user E, 30 from user H, and 120 from user K. Some of the low figures may reflect disappointment with the limitations of the interface; in particular, VALET's inability to recognize aliases may have reduced the number of commands that some users entered. However, the comments that users made on their questionnaires did not indicate wholesale dissatisfaction with the interface. Perhaps the volunteers, most of whom were students, simply had little work to

Table 4.2. Summary of Users' Inputs and Errors

User	Number of Inputs Accepted	Corrected	Rejected	Number of Errors Unintentional	Intentional
A	499	7	23	18	
B	153	12	11	10	
C	68	12	6	3	7
D	56	21	9	3	24
E	73		3	2	
F	56		9	4	
G	36	2	6	4	
H	15	2	4	2	
I	16	1			1
J	13	3	1		2
K	9				
Total	994	60	72	46	34

do during the summer, VALET's testing period, when classes were out of session.

Although VALET's testers as a group entered fewer commands than expected, a substantial amount of data was nevertheless collected during the user testing experiment. The 1,126 recorded commands represent a considerable amount of user interaction. Furthermore, VALET's recorded inputs are similar in many ways to those recorded in other user behavior studies [12]. With certain exceptions (described in Section 4.2.1), when compared to the results of other research efforts, VALET's users seem to have been typical: They invoked an ordinary assortment of UNIX commands and made the standard kinds of errors. In summary, the 1,126 commands recorded by VALET appear to be reasonably representative of most users' shell commands, so it is possible to analyze VALET's users' commands in order to evaluate VALET's effectiveness in "real world" situations.

The leftmost section of Table 4.1 describes VALET's handling of the complete set of recorded inputs. The middle two sections of the table, however, show how VALET responded to two disjoint subsets of those inputs: the set of all correct commands and the set of all incorrect commands. Errors come in many varieties, but for the purpose of the analysis presented here, the distinction between correct and incorrect commands is derived from the types of input errors that VALET was designed to recognize and correct. In particular, a command is considered to be correct if the entered command name corresponds to an actual command and the command arguments follow the required command syntax and correctly refer to entities of the required types. A command is considered to be erroneous, then, if it does not meet the above criteria for correctness. This means that a command is considered to be erroneous if, for example:

- the first word of the command does not name an actual shell command;
- the command arguments do not follow the required syntax for the command; or
- any of the command arguments do not properly name objects of the required types.

In other words, the analysis presented below is based on errors of *expression*, not errors of *intent*. The set of erroneous commands does not include commands that are well-formed but which in the context of a user's transcript appear not to be what the user perhaps meant to enter. For example, if a user entered the command "`ls`" in order to list the contents in the current directory and then immediately entered "`ls -a`" in order to see the names of the "dot files" that were omitted from the first listing, the original "`ls`" command would not be considered erroneous. Perhaps the user meant to enter "`ls -a`" in the first place, but "`ls`" itself is a valid command and without detailed knowledge of the user's task there is no reason to prefer one of these commands over the other. On the other hand, in the analysis presented here, correctly formed shell inputs are considered to be correct even if VALET is unable to interpret them. For example, the input "`!!`" which refers to the C shell's built-in command history is considered to be correct, even though VALET's parser cannot understand that input. Similarly, invocations of aliases are considered to be correct (when they are correctly invoked) even though VALET has no knowledge of its users' aliases, as explained previously in Section 3.4.4.

The VALET session transcripts were examined by hand in order to locate all the recorded erroneous inputs. As Table 4.1 indicates, 80 such inputs were found. Although the 11 recruited VALET users were asked not to make intentional errors while using the interface, from examination of the transcripts it is obvious that some of the users (especially user D) explored the system by entering various types of garbled shell commands. Since the purpose of the experiment was to demonstrate VALET's ability to process naturally occurring errors, not fabricated inputs, the set of erroneous commands was divided into two: those that were apparently unintentional and those that were apparently intentional, as determined by the context of the errors and the best judgment of the author of VALET. Intentional errors were not entirely eliminated from the analysis presented here, however, because although they were artificially constructed they are still actual errors that provide insight into VALET's capabilities. Many of the tables in this chapter therefore present data for both the set of all erroneous inputs and then for just the set of unintentionally erroneous inputs. The rightmost portion of Table 4.1 summarizes VALET's responses to its users' unintentionally erroneous inputs.

Because the 11 VALET test subjects were all experienced users of the UNIX C shell, it was naturally expected that the number of erroneous commands would be low as a percentage of all inputs. The actual error rate during the experiment, however, was surprisingly high. As stated above and shown in Table 4.1, 80 erroneous inputs were located in the VALET session transcripts. This means that 7.1% (about 1 in 14) of all the recorded inputs contained an error of the sort previously described: an incorrect command name, an incorrect command argument, a mislocated file name, or other recognizable input error. Even when only the 46 unintentional errors are considered the overall error rate remains relatively high at 4.2% — about 1 in 24. These figures suggest that the kinds of errors that VALET was meant to address actually occur at significant rates in shell inputs. Apparently there is a great deal of opportunity for a system such as VALET to improve the shell's command line interface.

That improvement, of course, depends on the system's ability to locate input errors — to distinguish between correct and incorrect commands. Table 4.1 shows that in general, VALET accurately made that distinction. An overwhelming percentage of all correct shell commands — 94.0% — were accepted by VALET and sent to the shell process verbatim. Furthermore, the great majority of all erroneous inputs were recognized by the interface and were withheld from the shell process. VALET offered corrections for 51.25% of all incorrect inputs and offered error messages for an additional 35%, meaning that over 86% of all erroneous inputs were at least recognized, and in most cases a correction was offered to the user. The accuracy of VALET's error recognition is practically unchanged when only the unintentionally erroneous inputs are considered, although in that domain VALET rejected a few more inputs than it corrected.

VALET was able to distinguish correct and incorrect inputs in most but not all cases. Not all actual input errors were recognized, and in some cases, VALET rejected or offered a correction for an already correct command. As Table 4.1 indicates, 63 truly correct inputs were misidentified as erroneous: 19 of these were corrected and the remaining 44 were rejected. These 63 misidentified inputs are the false positives resulting from VALET's error detection heuristics and constitute 6.0% of all actually correct inputs.

As explained in Sections 4.2.2 and 4.2.3 below, most of these false positives arose from users' attempts to employ parts of the shell input language that VALET was not designed to understand: in particular, command aliases and references to the C shell's built-in command history. Most of VALET's users ran into these limitations as indicated by users' written comments after the experiment. In general, VALET's testers wrote that VALET's intelligent features were useful but that the system's limitations were "discouraging." Comments such as this were typical: "I felt comfortable [with VALET], except for the fact that a large chunk of UNIX that I was used to wasn't really implemented yet. These were just convenience devices (like '`!!`')... but still I felt myself working against the system at times." Another user wrote that VALET's intelligent features were "somewhat useful" but that they "definitely didn't make up for the loss of aliases." The 11 VALET testers were warned about VALET's parsing restrictions, but clearly, users' established habits are hard to change.

In addition to false positives, the data from the experiment also revealed several false negatives: cases in which VALET failed to detect actual errors. Eleven truly erroneous commands were accepted by VALET as correct during the study. In practically all of these instances, however, VALET's failure to identify the input error was due to VALET's lack of knowledge about the invoked command, as explained later in Section 4.2.4. Without knowledge of the invoked command's required syntax, VALET was unable to correct errors that were present in the command line. Additional knowledge, therefore, would reduce the number of unrecognized input errors. Greater knowledge and improved correction heuristics would also improve the ratio of corrected to rejected inputs by enabling VALET to make reasonable corrections in a greater number of cases.

In summary, Table 4.1 shows that VALET responded appropriately to the great majority of all the shell commands that were entered during the user testing experiment. VALET in general accepted correct commands and corrected or rejected incorrect commands. The next three sections of this chapter more closely analyze the inputs within each of VALET's three response categories.

4.2.1 Analysis of All Accepted Inputs

VALET accepted 994 inputs during the user testing experiment. Of these, 983 (98.9%) were correct commands and only 11 (1.1%) were actually incorrect (i.e., unrecognized errors). As a percentage of all accepted inputs, therefore, the number of unrecognized errors is very low. The number of unrecognized errors is small when compared either to the total number of accepted inputs (994) or to the total number of erroneous inputs (80).

The high accuracy of VALET's parser is due in large part to VALET's knowledge of the most commonly invoked shell commands. As previously discussed in Section 3.4.4, VALET's shell command knowledge base contains detailed descriptions of approximately 50 commands, including many of the most frequently used commands such as `cd`, `ls`, `cp`, and `rm`. Each description provides information about the command line syntax required by a particular command, and when combined with VALET's other knowledge bases, these command definitions allow the intelligent shell interface to recognize and correct errors within invocations of the described commands. Ultimately, the effectiveness of VALET's ability to identify errors in command arguments is determined by both the number of commands for which VALET has detailed knowledge and the frequency at which each of those commands is invoked. The fact that VALET accurately distinguished between correct and incorrect inputs during the experiment — only 11 errors were unrecognized — suggests that VALET's built-in knowledge was applicable to most user inputs, and in fact the data from the users' transcripts support this conclusion. Table 4.3 shows that although VALET has detailed descriptions for only 50 or so commands, those descriptions applied to most of the commands that were entered by VALET's users. Overall, 57.5% of all the accepted, correct inputs entered during the user testing experiment were invocations of the 50 or so "defined" commands, commands for which VALET has explicit knowledge of the required command line syntax and arguments. "Generic" commands as described in Section 3.4.4, on the other hand, are those commands that are known to exist but for which VALET has no special knowledge.

When all of the accepted correct inputs are considered, VALET's built-in command

Table 4.3. Categorization of Accepted Correct Inputs

	Accepted Correct Inputs			
	From All Users		Excluding User A	
Category	#	%	#	%
Invocation of a defined command	565	57.5	405	83.0
Invocation of a generic command	418	42.5	83	17.0
Total	983	100.0	488	100.0

descriptions applied to 57.5% of all the accepted, correct inputs. This figure is low when compared to the results of other studies, however. For example, Hanson, Kraut, and Farber [12] studied the use of UNIX shell commands and found that the 20 most popular commands accounted for about 70% of all user inputs. In light of this result one would expect VALET's set of 50 command definitions to apply to more than 70% of its users' inputs, and in fact, the lower than expected applicability of VALET's command knowledge can be explained by the set of commands entered by user A. During the period of the VALET experiment, user A was developing a large computer program. User A created a special suite of scripts designed especially for this task, and not surprisingly he used these scripts quite frequently. In addition he used other program development tools that were not explicitly described within VALET. The effect was that many of user A's inputs referred to a small, unusual set of "generic" commands, and because user A entered almost half of all the inputs recorded during the VALET user testing experiment, his uncommon inputs had a very great effect on the statistics presented above.

When the inputs from user A are excluded, it becomes clear that VALET's built-in command descriptions applied to a great majority of its users' typical inputs. Table 4.3 shows that when the inputs from user A are ignored, 83.0% of the remaining accepted, correct inputs were invocations of commands explicitly described within VALET. Table 4.4 illustrates this fact in greater detail by listing the commands that were most frequently invoked during the user testing experiment. When inputs from all users are included, 8 of the 16 most popular commands are generic; when inputs from user A are excluded, only 3 of the top 16 are generic. The latter list is similar to the list of most popular commands determined by Hanson et al., which showed that most shell inputs are invocations of a relatively small set of orienting and data-acquisition commands (e.g., `ls` and `more`), general manipulation commands (e.g., `cd` and `rm`), and social commands (e.g., `mail`). Tables 4.3 and 4.4 make it clear that many of user A's recorded inputs were atypical. Overall, the results show that VALET's built-in command knowledge is in fact applicable to most users' typical shell inputs, and these results are consistent with the findings of other researchers.

The overall applicability of VALET's command knowledge resulted in accurate parsing of its users' inputs; as stated previously, 94.0% of all correct inputs were accepted by VALET and 86.25% of all erroneous inputs were recognized. Only 11 erroneous inputs were undetected by the interface, and these few instances are summarized in Table 4.5.

Table 4.4. Summary of the Most Frequently Invoked Commands

Accepted Correct Inputs					
From All Users			**Excluding User A**		
Command	**#**	**%**	**Command**	**#**	**%**
`ls`	169	17.2	`ls`	121	24.8
`cd`	127	12.9	`cd`	79	16.2
^{A}`tl`	70	7.1	`exit`	34	7.0
^{A}`build`	69	7.0	`rm`	32	6.6
^{G}`cc`	48	4.9	`more`	25	5.1
`rm`	44	4.5	`pwd`	23	4.7
`exit`	38	3.9	`cat`	11	2.3
^{A}`on`	29	3.0	`lpr`	11	2.3
`cat`	25	2.5	`finger`	10	2.0
`more`	25	2.5	^{G}`w`	10	2.0
^{G}`od`	24	2.4	^{G}`clear`	8	1.6
`pwd`	24	2.4	`mail`	7	1.4
^{A}`plccomp`	18	1.8	`ps`	7	1.4
^{A}`uclGrep`	17	1.7	`rlogin`	6	1.2
`ps`	16	1.6	`who`	6	1.2
^{A}`x`	16	1.6	^{G}`du`	5	1.0
Total	759	77.2	**Total**	395	80.9

A indicates a generic command created by user A.
G indicates a generic but standard UNIX command.

Table 4.5. Categorization of Accepted but Erroneous Inputs

Type of Error	**Instances**
Mistyped command name	5
Mislocated file	2
Argument not in required domains	1
Argument refers to wrong domain	1
Wrong number of arguments	1
Incompatible arguments	1
Total	11

Almost half of the undetected errors were typographical errors within a command name, and these errors were undetected because each mistake resulted in a valid command name — in particular, the name of a "generic" command about which VALET knew almost nothing. This meant that in each case, VALET could not use knowledge about the referenced command's required arguments in order to recognize its user's error. It should be noted, however, that three of the five mistyped command names were obvious, deliberate attempts to fool the interface.

In two other cases, a command line argument specified an incorrect location for a file. In other words, each file was referenced as if it were in a place other than its actual location. One of these errors was undetected because the invoked command was generic, and in the other case, VALET accepted the incorrect file name because it was confused about the shell process' current directory. Due to the implementation of the interface as separate processes (as described in Section 3.2), it is possible in rare circumstances for the GNU Emacs and Common Lisp components of VALET to lose track of the shell process' current directory. For example, if VALET were to misidentify a `cd` command as a non-shell input, based on the heuristics described in Section 3.3.1, the actual shell process would change its current directory without the knowledge of VALET's other components. Although rare, this kind of disorientation caused VALET to misinterpret six inputs during the experiment, and one of those inputs contained the mislocated file name noted above.

The remaining unidentified errors occurred in the arguments to generic commands. One argument was intentionally nonsensical; it did not name any object in the appropriate domain and it was not a mistyping of an acceptable name. In another case, an argument named a symbolic link when it was required to name a directory. The final two cases were input by user A: He invoked one of his personal scripts without its required argument, and he later invoked a compiler with incompatible arguments.

In summary, most of the 11 unrecognized errors described in Table 4.5 were missed because VALET lacked knowledge about certain shell commands. In all but one case, the user's error might have been detected or even corrected had VALET contained detailed knowledge of the invoked command.

4.2.2 Analysis of All Corrected Inputs

VALET offered corrections for 60 inputs during the user study, and these cases are summarized in Table 4.6. As shown, VALET made corrections for 41 erroneous inputs, including 24 mistyped command names and 1 incorrect command name. The difference

Table 4.6. Categorization of Corrected Inputs

Category	Corrected Inputs #	%
Actually Erroneous Inputs		
Mistyped command name	24	
Incorrect command name	1	
Mistyped command argument	15	
Mislocated file	1	
Subtotal	41	68.3
Actually Correct Inputs		
Invocation of alias	14	
Invocation of program in current directory	3	
Confusion about shell's current directory	2	
Subtotal	19	31.7
Total	60	100.0

between these classifications is that a mistyped command name is the result of an apparent typographical error, whereas an incorrect command name is not. (In the single case of an incorrect command name, VALET's user entered "`x -x`" when his apparent intent was actually "`ps -x`".) VALET also recognized and corrected 15 typographical errors in command arguments of various kinds and additionally corrected 1 incorrect reference to a file. In that instance, VALET's user entered "`cd par`" and VALET determined that "`cd ../par`" was most likely what the user intended. It was disappointing to discover that VALET corrected only 1 of the 9 mislocated file names that were entered during the user testing experiment; although 7 of these errors were recognized, VALET was unable to offer a reasonable correction for 6 of those errors. The apparent causes of these failures are described later in Section 4.2.4.

In addition to correcting 41 erroneous inputs, VALET mistakenly identified 19 truly correct commands as erroneous and offered corrections for them. As Table 4.6 shows, however, most of these 19 false positives arose from users' attempts to invoke their command aliases. Alias names are often short and lexically similar to the names of other commands. VALET, because it had no knowledge of its users' aliases (as described in Section 3.4.4), interpreted many alias names as input errors and therefore offered corrections for those commands. If it had been possible for VALET to determine its users' alias definitions, most (probably all) of these inappropriate corrections would have been eliminated.

Three more inappropriate corrections arose from users' attempts to invoke programs located in the shell's current directory. It is common for users of the C shell to put the directory "." in the shell's search path, which makes it possible for users to invoke programs residing in the shell's current directory simply by typing the names of those programs as commands. However, the current directory of the shell changes over time, and VALET scans the shell's search path only once for each user for reasons previously described in Section 3.4.4. Because of this limitation, VALET does not understand the effect of placing "." in the shell's path and so does not understand the just-described method of program invocation. Users were warned of this restriction and were told how to work around it,[1] but established habits are difficult to change.

Finally, two inappropriate corrections were made because VALET's Common Lisp component had inaccurate data about the shell process' current directory. In total, only a very small number of truly correct inputs were misidentified as erroneous by the interface. The 19 misidentified inputs listed in Table 4.6 amount to just 1.8% of the 1,046 correct inputs that were recorded during the user testing experiment.

4.2.3 Analysis of All Rejected Inputs

Whenever VALET detected an error in an input command but could not offer any reasonable correction for that error, VALET simply rejected the entire input command and displayed an appropriate explanation of the problem to its user. During the user testing experiment VALET rejected 72 inputs, and these inputs are categorized in Table 4.7. Of the 72 rejected inputs, 28 (38.9%) were truly incorrect and 44 (61.1%) were truly correct but misinterpreted by the interface.

Most of the 28 rejected erroneous inputs fall into the categories described previously: mistyped command names and arguments, mislocated files, and arguments that do not have any obvious interpretations or corrections within their required domains. Although all of these errors were detected, none of them were corrected by the interface. In some cases this was due to a lack of contextual information — for example, each of

[1] Within VALET, programs in the shell's current directory can be invoked by prefixing the program name with "./". This is the syntax normally required by the C shell when "." is not in the shell's command search path.

Table 4.7. Categorization of Rejected Inputs

Category	Rejected Inputs #	Rejected Inputs %
Actually Erroneous Inputs		
Mistyped command name	5	
Mistyped command argument	4	
Mislocated file	6	
Argument not in required domains	2	
Argument refers to wrong domain	1	
Missing “required option”	1	
Program not in search path	1	
No match for glob	2	
Whitespace error	1	
Nonsensical input	5	
Subtotal	28	38.9
Actually Correct Inputs		
Invocation of alias	5	
Invocation of undefined built-in command	8	
Invocation of program in current directory	7	
Use of command history (`!`)	11	
Use of shell variable (`$`)	1	
Use of pipeline (`|`)	5	
Confusion about shell's current directory	3	
Bug in VALET's expansion of globs	4	
Subtotal	44	61.1
Total	72	100.0

the six file location errors was uncorrected because VALET's file system knowledge base had not yet scanned (or rescanned) certain portions of the actual file system, and the correction heuristics were purposely prevented from updating VALET's internal file system model. (The file name correction heuristics are described in Section 3.4.5.3.) In other circumstances, VALET was simply unable to locate an appropriate, sufficiently similar, correct alternative for a mistyped term. Many of the uncorrected errors provide insight into ways in which VALET's correction heuristics could be improved, as described later in Section 4.2.4.

In addition to the kinds of erroneous inputs just described, VALET rejected a handful of inputs that were instances of other miscellaneous kinds of errors. For example, one VALET user attempted to remove a directory with an `rm` command and apparently forgot that `rm` will remove a directory only when it receives the "`-R`" option in addition to the directory name. VALET told that user that he had forgotten to use the "`-R`" option. VALET could have offered the obvious correction in that instance, but the existing transition network for `rm` (shown previously in Figure 3.7) was not designed to do so. Other instances of errors that VALET was not designed to correct include these: one invocation of a program not in the shell's search path (and not explicitly described within VALET), two file name patterns that did not match the names of any actual files, one case of conjoined terms (in which the user typed "`/etc/ping/asylum`" instead of "`/etc/ping asylum`"), and five inputs that were uninterpretable as shell commands. Two of the nonsensical inputs were apparently intentionally typed random strings and the other three were actually lines of hexadecimal output from the UNIX `od` program!

In addition to rejecting 28 erroneous inputs, VALET also rejected 44 well-formed shell commands. As Table 4.7 shows, however, the great majority of these rejected but correct inputs were misunderstood because they made use of language features not known to the interface. Of the 44 correct but rejected inputs, 17 relied upon unimplemented syntactic features — history references, shell variables, and command pipelines — and 20 were attempts to invoke commands that were either unknown to the interface (e.g., aliases) or located in the shell's current directory. Due to an oversight, some of the C shell's built-in commands were not described to VALET, and this lack of knowledge caused 8 inputs that should have been parsed to instead be rejected. Finally, 7 inputs were rejected due to temporary problems within the interface. Confusion about the shell's current directory (as described in Section 4.2.1) caused three inappropriate rejections and a programming error in VALET's input tokenizer caused VALET to misinterpret four file name patterns and subsequently reject the expanded commands.[2]

VALET rejected or corrected 63 truly correct shell commands during the user testing experiment, and almost all of these cases arose from the intentionally chosen limitations of VALET's current implementation. Although these false positives amount to just 6.0% of all the correct commands recorded during the study, at the conclusion of the experiment many of VALET's testers complained about the system's inability to understand aliases and the other syntactic shortcuts provided by the standard C shell. One user wrote: "The usefulness of the [VALET] shell was a tradeoff between valid correction of errors and frustration at not being allowed my favorite aliases. I had not realized how much I relied upon previously defined aliases until this experiment." Several other users expressed similar sentiments (as noted in Section 4.2), so it is apparent that the limitations of VALET's current implementation can noticeably hinder experienced users of the C shell. However, these shortcomings are the result of VALET's experimental nature and design, and in spite of its limitations VALET served its purpose and to a great extent met its goal of providing an "intelligent" interface to the UNIX C shell. The data gathered from the user testing study demonstrate that on the whole, VALET accurately distinguished between correct and incorrect inputs and that VALET offered reasonable corrections for its users' most frequent input errors.

4.2.4 Analysis of All Erroneous Inputs

Table 4.8 shows that the errors made by VALET's testers were largely those that VALET was tailored to recognize and correct. Most of the recorded erroneous inputs resulted from typographical slips, and by far, most of those slips were isolated "simple" errors: the insertion, deletion, or substitution of a single character, or the transposition of two adjacent characters within an input term. Only three unintentional typographical errors demonstrated more serious mutations. (In one interesting case, one of VALET's users apparently misplaced his hand on the computer keyboard and typed "`xs`" when he apparently meant to type "`cd`". VALET suggested "`ls`" as a correction.) The fact that most typographical errors were simple is consistent with the results of other studies

[2]The problem was that VALET's tokenizer included the names of VALET's internal "imaginary" directory entries in the expansions of file name patterns! (Imaginary entries are described in Section 3.4.5.1.) This problem was corrected immediately once it was discovered, shortly after the start of the user testing experiment.

Table 4.8. Categorization of Erroneous Inputs by Type

Category	All Erroneous Inputs #	%	Unintent'ly Erroneous Inputs #	%
Typographical Errors				
Simple	38		20	
Complex	15		3	
Subtotal	53	66.25	23	50.0
File Mislocation				
Extra directory components	1		1	
Missing directory components	7		6	
Wrong directory components	1		1	
Subtotal	9	11.25	8	17.4
Incorrect Command Arguments				
Argument not in required domains	3		2	
Argument refers to wrong domain	2		2	
Wrong number of arguments	1		1	
Incompatible arguments	1		1	
Missing "required option"	1		1	
Subtotal	8	10.00	7	15.2
Other Errors				
Incorrect command name	1		1	
Program not in search path	1		1	
No match for glob	2		2	
Whitespace error	1		1	
Nonsensical input	5		3	
Subtotal	10	12.50	8	17.4
Total	80	100.00	46	100.0

that characterize users' typographical errors [5, 6, 11, 28]. In total, typographical errors were by far the most common type of erroneous input recorded during the VALET study, accounting for 66.25% of all errors and 50.0% of all unintentional errors. VALET was tailored to correct these kinds of errors and actually did so, within the limits imposed by its knowledge bases.

VALET was also intended to correct file location errors, and this type of error actually arose with significant frequency during the study: 17.4% of all unintentional errors were file location errors. In six of the nine recorded mislocations, VALET's user omitted a single directory component from the required file name, and in all but one of those inputs it was the first component that was missing. (Often, the incorrectly referenced file was actually either one level above or below the shell's current directory in the file system hierarchy.) In one case a user omitted two directory components and in the remaining mislocated file names VALET's user either inserted an extra component or specified an incorrect component. Clearly, "spatial" context is important for correcting mislocated file names, and VALET's heuristics as described in Section 3.4.5.3 make use of that information. (However, the restrictions placed upon these heuristics caused VALET not to correct many file location errors, as explained below.) None of the recorded file location errors were combined with typographical errors, although VALET was designed to handle such situations.

The remaining types of error listed in Table 4.8 are types that VALET was designed to detect but for which VALET has no specific correction procedures. VALET's treatment of these errors is therefore not surprising: Although more than three-quarters of these remaining errors were recognized (as previously listed in Table 4.7) by the interface, VALET did not offer a correction for any of these mistakes. Many of these errors defy automatic correction because they provide no useful information about the user's intent; for example, when an entered command argument is completely unlike any acceptable argument, VALET has no lexical context from which to make meaningful inferences. Some of the recognized but uncorrected errors possibly could have been corrected if VALET had contained special knowledge about its users' tasks. (SAUCI, the shell interface described in Section 2.4, incorporates this kind of knowledge for two very specific domains.) Other recognized but uncorrected errors could have been processed had VALET contained a wider array of correction heuristics. It would be interesting to add a special heuristic for correcting whitespace errors, for instance, or one for correcting file name patterns.

Those kinds of special-purpose correction procedures, however, would be in general less useful that the procedures that VALET already contains, which enabled VALET to make reasonable corrections for most of the input errors recorded during the experiment.

Table 4.9 summarizes VALET's performance in detecting and correcting errors. VALET offered a correction for 51.25% of all the erroneous inputs that were recorded, and in about two-thirds of those cases VALET's correction was accepted verbatim by VALET's user. In other words, in most of the cases in which VALET offered a correction for an erroneous input, the user's next command was *exactly* the command that VALET had just suggested. This is true even when only the set of unintentionally erroneous inputs is considered. (In that smaller domain, however, VALET offered corrections for a somewhat smaller percentage — 39.1% — of users' errors.) It appears, therefore, that VALET's corrections were frequently appropriate and useful. Most of VALET's users agreed. At the end of the experiment, most indicated that VALET could often but not always detect and correct their most common mistakes. One user wrote: "[The] mistakes I made (usually simple typo errors in command names) were speedily picked up by VALET.... My typos were those (for the most part, say 90% of the time) which VALET could correct."

Although VALET apparently often discerned its users' intentions, not all of VALET's suggested corrections were confirmed by users. In a few cases VALET's user changed the suggested command name or arguments to lexically similar alternatives or added additional arguments to the corrected command. In other cases, VALET's user discarded the interface's suggestions entirely and entered a completely new command, apparently because the user had changed his or her mind about what command to enter next. Many of the cases in which a user appeared to change his or her mind, however, were actually due to intentional experimentation upon the interface. A few people tested VALET by purposely entering a variety of incorrect commands in order to discover the capabilities of the system. In only two cases did VALET's correction of an unintentional error cause a user to completely change course.

About half of all erroneous inputs were corrected, which unfortunately means that about half were not. Out of 80 erroneous inputs, 28 were simply rejected (with explanation, but without correction) and 11 were mistakenly accepted as correct by the interface. The reasons for which VALET failed to correct these inputs are listed in Table 4.10. In general, whenever VALET rejected an erroneous input (i.e., correctly detected an input error but failed to offer any correction for that error), the failure was due to one of

Table 4.9. Categorization of Erroneous Inputs by Outcome

Category	All Erroneous Inputs #	All Erroneous Inputs %	Unintent'ly Erroneous Inputs #	Unintent'ly Erroneous Inputs %
Correction Offered to User, and User Accepted Correction Verbatim				
VALET corrected command name	17		7	
VALET corrected command arguments	8		5	
Subtotal	25	31.25	12	26.1
Correction Offered to User, but User Did Not Accept Correction Verbatim				
User changed command name	2		2	
User changed command arguments	3		2	
User changed his or her mind	11		2	
Subtotal	16	20.00	6	13.0
Error Recognized, but No Correction Offered to User				
VALET rejected erroneous command	28		21	
Subtotal	28	35.00	21	45.7
Error Not Recognized				
VALET accepted erroneous command	11		7	
Subtotal	11	13.75	7	15.2
Total	80	100.00	46	100.0

Table 4.10. Categorization of Uncorrected Erroneous Inputs

Category	All Erroneous Inputs #	%	Unintent'ly Erroneous Inputs #	%
Error Recognized, but No Correction Offered to User				
Uncorrected due to lack of context:				
Mistyped name of known command	1			
Mistyped name of unknown command	3		3	
Mislocated file	6		6	
No lexically similar alternative found for:				
Mistyped name of known command	1			
Mistyped command argument	4		1	
Incorrect (not mistyped) argument	2		2	
Program not in search path	1		1	
Argument referred to wrong kind of file	1		1	
Missing "required option"	1		1	
No match for glob	2		2	
Whitespace error	1		1	
Nonsensical input	5		3	
Subtotal	28	71.8	21	75.0
Error Not Recognized, so Erroneous Input Accepted				
Lack of knowledge about command	10		6	
Confusion about shell's current directory	1		1	
Subtotal	11	28.2	7	25.0
Total	39	100.0	28	100.0

two situations: Either VALET lacked the contextual information that would have allowed the interface to make a correction, or VALET had the necessary context but its spelling correction heuristics were simply unable to find a sufficiently lexically similar alternative to the mistyped term. Just as missing knowledge sometimes prevented a correction, it sometimes caused VALET to accept erroneous inputs. The bottom portion of Table 4.10 shows that in all but one case, each time VALET missed an actual input error, the reason was that VALET lacked knowledge about the command being invoked.

Lack of knowledge accounts for about half of all the cases in which VALET failed to correct an error. (This is true even when only unintentional errors are considered.) Ten of the 11 unrecognized input errors were missed because those inputs referred to generic commands, commands for which VALET had no knowledge of the required command line syntax and arguments. In five of those cases (listed previously in Table 4.5) a typographical error in a command name transformed the intended name into the name of another command. In each case, however, the error transformed the intended command name into the name of a generic command, so VALET was unable to use knowledge about the entered command line arguments in order to detect the user's input error. In the five other cases, errors within the arguments given to generic commands went unnoticed due to the corresponding gaps in VALET's knowledge. Finally, inaccurate information about the shell's true current directory prevented the interface from recognizing one mislocated file name.

The other cases in which a lack of knowledge prevented VALET from issuing a correction are listed under the "Uncorrected due to lack of context" heading in Table 4.10. One user intentionally mistyped the name of a known (but generic) command. VALET failed to offer the obvious correction because that generic command had never before been invoked by that user, and VALET's command corrector does not consider the name of a generic command to be a candidate correction until that command has been invoked at least once. In three other cases users mistyped the names of aliases or undefined built-in shell commands and VALET failed to determine the appropriate corrections because it had no knowledge of the users' intended commands. Finally, in six cases, VALET's file name correction procedures failed because VALET's file system knowledge base did not contain up-to-date data about certain parts of the actual file system.

The failures of the file name correction procedures arose primarily because VALET constrained those procedures and prevented them from updating VALET's internal file system model in certain situations. These constraints were implemented in order to increase the speed of the corrector (as described in Section 3.4.5.3), but in retrospect it is clear that the restrictions were too severe and too greatly diminished VALET's ability to correct mislocated file names. For example, one of VALET's users entered the command "`uncompress Intro.ps.Z`" when in fact the file "`Intro.ps.Z`" had just been created within the directory `atrium`, a child of the shell's current directory. Although VALET recognized that the entered file name was not valid, the interface was unable to make the appropriate correction for the following reasons. VALET's knowledge of the `atrium` directory was out of date; the modeled contents of that directory had not been updated to include the just-created "`Intro.ps.Z`" file. When VALET's parser discovered that the input file name was invalid, it invoked the procedure described in Section 3.4.5.3 to search the "neighborhood" of the shell's current directory for a file of the given name. Unfortunately, that search was restricted: It was not allowed to rescan the contents of any of the children of the shell's current directory. (Because the number of children can be very large, rescanning all of the children of a directory can be very time consuming.) Although VALET could have discovered the "`Intro.ps.Z`" file by updating its model of the `atrium` directory, the corrector was barred from doing so, and the result was that VALET failed to offer the appropriate correction to its user. This same restriction prevented VALET from updating its file system model in other cases as well, each time preventing VALET from acquiring the context needed in order for it to make an appropriate correction.

The data gathered from the user testing experiment show that the effectiveness of VALET's input correction heuristics could be increased by improving VALET's knowledge. VALET would benefit from new, fixed kinds of knowledge (e.g., descriptions of additional commands), and it would also benefit from fine-tuning of its ability to keep its existing knowledge up to date. The interface would also profit from improved spelling correction algorithms. As shown in Table 4.10, in a small but significant number of situations VALET had all the context it required but was still unable to discover an appropriate, sufficiently similar alternative for a mistyped term. For example, VALET failed to suggest "`whoami`" for the intentionally mistyped input "`whoajsi`" although the two terms are quite similar. In another instance, VALET failed to suggest "`Intro.ps`" as a correction for the unintentionally erroneous file name "`Intro.ps.Z`". (The "`Intro.ps.Z`" file was mentioned in the previous paragraph. That file was decompressed to create "`Intro.ps`" and the original "`Intro.ps.Z`" file was deleted. After those changes had occurred, however, VALET's user entered the original "`Intro.ps.Z`" file name in a subsequent command.) In each of these cases VALET refused to offer the appropriate correction because the similarity of the input to the correct term (as measured by the functions `lexicon-spell` and `lexicon-guess` described in Section 3.4.3) fell below an arbitrary predetermined threshold. The examples just described from the user testing experiment suggest that VALET's spelling correction heuristics and thresholds could be modified in order to better handle certain kinds of errors. Combined with additional knowledge and improved knowledge maintenance, such changes could have allowed VALET to correct, rather than simply detect, a significant number of the errors recorded during the user testing experiment.

The preceding paragraphs describe how VALET detected and "came close" to correcting certain errors. Although it is important to understand how VALET could be improved, it is also important to realize that the data from the user testing experiment demonstrate that VALET is already a reasonably "intelligent" interface to the UNIX C shell. Despite its various limitations, VALET accurately distinguished most correct and incorrect inputs during the user testing experiment. Experienced users of the C shell appear to make input mistakes with significant frequency, and the kinds of mistakes that such people make most often are those that VALET was designed to detect and correct. For about half of all the recorded erroneous inputs VALET offered a reasonable correction to its user, and as summarized in Table 4.11 most of these corrections were accepted verbatim and immediately reinput to the interface. Roughly two-thirds of VALET's corrections to erroneous inputs were accepted verbatim, which demonstrates that to a very significant degree, VALET was actually useful to its users. VALET was able to knowledgeably infer its users' intentions in order to correct many of their input errors, thereby providing a user-supportive and "intelligent" interface to the C shell.

Table 4.11. Categorization of Corrected Erroneous Inputs

	All Erroneous Inputs		**Unintent'ly Erroneous Inputs**	
Category	#	%	#	%
VALET's correction was accepted verbatim	25	61.0	12	66.7
VALET's correction was not accepted verbatim	16	39.0	6	33.3
Total	41	100.0	18	100.0

CHAPTER 5

CONCLUSION

The results of the experiment described in Chapter 4 show that VALET has both important strengths and weaknesses. VALET proved that through knowledge and context it could accurately detect most of the errors in its users' commands. The errors made most often by experienced users of the C shell appear to be those that VALET was designed to recognize and correct, and furthermore, when VALET offered corrections for erroneous commands, VALET's users most often accepted the suggestions offered by the interface. These results suggest that in general, a command line interface that uses knowledge in order to flexibly interpret its users' commands can also accurately detect and correct input errors. Such "intelligent" human-computer interfaces can be more cooperative and more user-friendly than their "unintelligent" counterparts.

The user testing experiment also highlighted some of VALET's shortcomings. Just as VALET's strengths derive from its incorporated knowledge, most of VALET's weaknesses derive from certain gaps in its knowledge. VALET refused to accept a significant number of well-formed shell inputs because it did not understand such things as user-defined aliases and references to the C shell's built-in command history. In addition, a significant number of actual errors were undetected because VALET had no detailed knowledge of certain commands. Finally, in some cases, VALET's out-of-date knowledge prevented it from making what should have been obvious and easy to determine corrections.

The most obvious way in which VALET could be improved, therefore, would be to increase the amount of knowledge within the system and also the accuracy of that knowledge. The most troublesome informational gaps in VALET were caused by the separation between the intelligent command parser (currently implemented in Common Lisp) and the actual C shell program. A result of VALET's prototypical design, this separation kept VALET from consulting information that was internal to the shell process, and subsequently, this lack of knowledge made it practically impossible for VALET to parse such things as aliases and shell variable references. Removing the division between

VALET's intelligent Common Lisp components and the actual shell would remove these barriers to knowledge. It would be possible and very useful, for example, to reimplement VALET's Common Lisp components in the C programming language and then integrate those components directly with the `csh` (or `tcsh` [27] or even `zsh` [7]) program. This integration would provide the intelligent command parser with access to the shell's internal data and would eliminate all of the problems that caused the current VALET implementation to mistakenly reject or correct truly well-formed commands. Integration and reimplementation in C would also greatly increase the speed of the system.

Even if it were integrated with the actual shell program, VALET would still need to contain its own descriptions of the other UNIX programs that are available. VALET's shell command knowledge base describes in detail only a small fraction of the hundreds of programs actually available to VALET's users, and during the user testing experiment a number of input errors were unrecognized simply because VALET had no special knowledge of the programs being invoked. VALET therefore would be improved if it contained detailed descriptions for a much wider assortment of programs. It would be interesting for VALET to attempt to "learn" about programs for which it has no explicit knowledge. Ideally, however, an intelligent UNIX shell would be able to determine the command line syntax and arguments required by a program by consulting the program itself. That approach would eliminate the need for the shell to have its own built-in (and therefore, possibly inaccurate) data about other programs' expected arguments. Unfortunately, there is currently no standard way for UNIX programs to communicate their command line requirements to the shell — and even if such a mechanism were invented, hundreds of existing UNIX programs would need to be changed in order to adopt the convention. Despite these barriers, however, it would be very interesting to research ways in which individual programs could communicate with command shells in order to make computer systems more user-friendly.

In addition to new command descriptions, VALET could also benefit from completely new kinds of knowledge. For example, one could incorporate user models into VALET. User models like those contained in SUSI [16] (described in Section 2.5) could describe which UNIX concepts are understood and which are not. It would perhaps be useful for VALET to understand English synonyms for certain UNIX commands; this ability might benefit inexperienced users and could even allow VALET to be used as a kind of UNIX training tool. It would also be interesting to explore how task-specific knowledge like that built into SAUCI [35] could be added to VALET. The kinds of information contained in SAUCI and SUSI would need to be expanded in order to be most useful in a general-purpose command shell such as VALET. Balancing these new kinds of knowledge with VALET's existing knowledge and input correction heuristics would be challenging.

VALET's input correction procedures could also be improved. As described in Section 4.2.4, in a small number of cases VALET had all the information it required but was nonetheless unable to find a sufficiently similar correction for an invalid input term. VALET's spelling corrector could be changed in order to handle some of those recorded inputs. It would also be possible to modify the spelling corrector to consult "character confusion matrices" or other data [11, 18] in order to rank candidate corrections according to the likelihoods of various keyboarding errors. VALET's input correction facilities could be improved in other ways as well. As explained in Section 4.2.4, the restrictions upon the file name correction heuristics need to be reduced and the various command definitions need to be improved in order to allow VALET to offer corrections for more classes of error. Most likely, this would involve defining some new parser actions in order to insert, delete, or rearrange input tokens. It would also be useful for the parser and its actions to evaluate the natures of input errors and their likely corrections in greater detail. VALET could then in some cases submit corrected inputs for execution without the need for confirmation by VALET's user. The need for confirmation would be determined by the seriousness of the located error, the likelihood that VALET's inferred correction is truly the proper correction, and the risk involved in submitting the corrected command without confirmation from the user. The parser and its actions would need to be changed in order to compute heuristic measures for each of these attributes.

Other changes could be made to the parser as well. The current parsing scheme has the disadvantage that ordinarily, once an erroneous token is found, the remainder of the input is ignored. (Input commands are generally parsed from left to right.) Unfortunately, the remainder of the command can often provide valuable information about the user's intent. VALET's current parser actions, therefore, sometimes delay the reporting of errors — in other words, upon recognizing an error they sometimes pretend that no error was found so that the remainder of the command may be parsed. Later, after all of the input has been processed, a special parser action determines if a previously unreported error should be signaled. By delaying parsing failures until the entire input command has been parsed, VALET can sometimes produce more accurate explanations of parsing failures.

(The generation of error messages is described in Section 3.4.2.3.) However, the need to employ tricks such as this indicates that a better parsing scheme could be devised. It would be interesting to explore other techniques (e.g., "best-first" ATN parsing) for analyzing VALET's inputs.

VALET's purpose was to demonstrate the effectiveness of knowledge-based "intelligent" interfaces, so it was designed to change only the way in which shell commands were parsed. Notably, VALET did not make any significant changes to the way in which information was *presented* to the shell's users. It is clear from the results of other studies [10, 12, 26, 35], however, that the C shell's user interface could be greatly improved through such changes. For instance, it would be very useful for the shell to display several separate windows of information. One window could contain the normal terminal-like session transcript and additional windows could provide contextual information such as the contents of the shell's current directory or the list of the user's most recently or frequently entered commands. VALET's existing GNU Emacs interface could be readily adapted in order to explore these new interface styles.

Finally, no matter what changes or improvements are made to VALET in the future, continued testing of the interface is a necessity. Only through actual use can the effectiveness of a human-computer interface be measured. The results of behavioral experiments can illustrate both the shortcomings of today's computer interfaces and the most effective methods for overcoming those problems in future systems.

Many of today's human-computer interfaces are difficult for people to use. In order to correct this situation, interfaces of the future will need to make significant efforts to meet the needs of their users. A large part of that task will be simply to understand users' intentions. VALET demonstrates that it is both feasible and profitable for command line interfaces to be "cooperative" and "intelligent" and tolerant of human error. The ideas embodied by VALET are valuable and worthy of incorporation into other human-computer interfaces because ultimately, intelligence is the quality that will characterize the most user-friendly and popular computer systems of tomorrow.

REFERENCES

[1] Benbasat, I., and Wand, Y. Command abbreviation behavior in human-computer interaction. *Commun. ACM 27*, 4 (Apr. 1984), 376–383.

[2] Bertino, E. Design issues in interactive user interfaces. *Interf. in Comput. 3*, 1 (Feb. 1985), 37–53.

[3] Card, S. K. Human factors and artificial intelligence. In *Intelligent Interfaces: Theory, Research and Design*, P. A. Hancock and M. H. Chignell, Eds. Elsevier Science Publishers B.V., Amsterdam, 1989.

[4] Charniak, E., and McDermott, D. V. *Introduction to Artificial Intelligence.* Addison-Wesley, Reading, Mass., 1985.

[5] Damerau, F. J. A technique for computer detection and correction of spelling errors. *Commun. ACM 7*, 3 (Mar. 1964), 171–176.

[6] Durham, I., Lamb, D. A., and Saxe, J. B. Spelling correction in user interfaces. *Commun. ACM 26*, 10 (Oct. 1983), 764–773.

[7] Falstad, P., Wischnowsky, S., Stephenson, P., et al. The Z shell (`zsh`), version 2.5. Source code and documentation available via anonymous FTP from `ftp.sterling.com:/zsh`, Dec. 1993.

[8] Gabriel, R. P., and Steele Jr., G. L. Editorial: What computers can't do (and why). *Lisp and Symb. Comput. 1*, 3/4 (Jan. 1989), 221–225.

[9] Good, M. D., Whiteside, J. A., Wixon, D. R., and Jones, S. J. Building a user-derived interface. *Commun. ACM 27*, 10 (Oct. 1984), 1032–1043.

[10] Greenberg, S., and Witten, I. H. How users repeat their actions on computers: Principles for design of history mechanisms. In *CHI '88 Conference Proceedings: Human Factors in Computing Systems* (May 1988), E. Soloway, D. Frye, and S. B. Sheppard, Eds., ACM, pp. 171–178.

[11] Grudin, J. T. Error patterns in novice and skilled transcription typing. In *Cognitive Aspects of Skilled Typewriting*, W. E. Cooper, Ed. Springer-Verlag, New York, 1983.

[12] Hanson, S. J., Kraut, R. E., and Farber, J. M. Interface design and multivariate analysis of UNIX command use. *ACM Trans. Office Inf. Syst. 2*, 1 (Mar. 1984), 42–57.

[13] Hawley, M. J. Interactive spelling correction in UNIX: The METRIC library. Tech. Mem. TM82–11221–20, Bell Laboratories, Aug. 1982.

[14] Hayes, P., Ball, E., and Reddy, R. Breaking the man-machine communication barrier. *Computer 14*, 3 (Mar. 1981), 19–30.

[15] Hayes, P., and Reddy, R. An anatomy of graceful interaction in spoken and written man-machine communication. Tech. Rep. CMU–CS–79–144, Carnegie-Mellon Univ., Pittsburgh, Pa., Sept. 1979.

[16] Jerrams-Smith, J. An attempt to incorporate expertise about users into an intelligent interface for UNIX. *Int. J. Man-Machine Stud. 31*, 3 (Sept. 1989), 269–292.

[17] Joy, W. An introduction to the C shell. In *UNIX User's Manual Supplementary Documents*. Univ. of California, Berkeley, Calif., Apr. 1986.

[18] Kukich, K. Techniques for automatically correcting words in text. *ACM Comput. Surv. 24*, 4 (Dec. 1992), 377–439.

[19] Ledgard, H., Whiteside, J. A., Singer, A., and Seymour, W. The natural language of interactive systems. *Commun. ACM 23*, 10 (Oct. 1980), 556–563.

[20] Lewis, B., LaLiberte, D., and the GNU Manual Group. *The GNU Emacs Lisp Reference Manual*, 1.02 ed. The Free Software Foundation, Cambridge, Mass., June 1990.

[21] McMillan, T. C., and Moran, B. P. Command line structure and dynamic processing of abbreviations in dialogue management. *Interf. in Comput. 3*, 3/4 (Sept. 1985), 249–257.

[22] Minasi, M. Building a smarter interface. *AI Expert 6*, 4 (Apr. 1991), 15–23.

[23] Morgan, H. L. Spelling correction in systems programs. *Commun. ACM 13*, 2 (Feb. 1970), 90–94.

[24] Nickerson, R. S. Why interactive computer systems are sometimes not used by people who might benefit from them. *Int. J. Man-Machine Stud. 15*, 4 (Nov. 1981), 469–483.

[25] Norman, D. A. The trouble with UNIX. *Datamation 27*, 12 (Nov. 1981), 139–150.

[26] Norman, D. A. Design rules based on analyses of human error. *Commun. ACM 26*, 4 (Apr. 1983), 254–258.

[27] Placeway, P., Zoulas, C., et al. The `tcsh` command shell, version 6.05. Source code and documentation available via anonymous FTP from `ftp.uu.net:/pub/shells/tcsh`, June 1994.

[28] Pollock, J. J., and Zamora, A. Automatic spelling correction in scientific and scholarly text. *Commun. ACM 27*, 4 (Apr. 1984), 358–368.

[29] So, B., and Travis, L. A step toward an intelligent UNIX help system: Knowledge representation of UNIX utilities. Tech. Rep. 1230, Univ. of Wisconsin-Madison, Madison, Wisc., Apr. 1994.

[30] Stallman, R. M. *GNU Emacs Manual*, 6th ed. The Free Software Foundation, Cambridge, Mass., Mar. 1987.

[31] Steele Jr., G. L. *Common Lisp: The Language*, 2nd ed. Digital Press, Bedford, Mass., 1990.

[32] Teitelman, W. *INTERLISP Reference Manual.* Xerox Palo Alto Research Center, Palo Alto, Calif., Dec. 1975.

[33] Teitelman, W., and Masinter, L. The Interlisp programming environment. *Computer 14*, 4 (Apr. 1981), 25–33.

[34] Tyler, S. W. SAUCI: A knowledge-based interface architecture. In *CHI '88 Conference Proceedings: Human Factors in Computing Systems* (May 1988), E. Soloway, D. Frye, and S. B. Sheppard, Eds., ACM, pp. 235–240.

[35] Tyler, S. W., and Treu, S. An interface architecture to provide adaptive task-specific context for the user. *Int. J. Man-Machine Stud. 30*, 3 (Mar. 1989), 303–327.