Design, Implementation, and Evaluation of Automatic Spelling Correction for UNIX Commands

J. David Hobbs & Ben Shneiderman*
Department of Computer Science
University of Maryland

Abstract

A UNIX shell (csh) was modified to automatically correct misspelled command lines. The design and implementation were not easy, contrary to the opinions of previous researchers. After implementation, the shell was given to 21 users for evaluation. Their comments and performance suggest that any change to a familiar system may overshadow potential benefits of a new system. Designers of spelling correctors must be extremely cautious in order to produce a satisfactory system. Minimal distraction from the user's task should be a prime goal in designers of correction facilities. Spelling corrector filters and methods of testing filters are discussed.

Acknowledgements

I would like to thank several people for help with this project. Chris Torek gave helpful advice on UNIX programming. Conversations with William Mosteller were helpful in understanding general spelling correction issues. Many people in the University of Maryland Electrical Engineering Department, Computer Science Department and Systems Research Center gave advice and tried out the shell. Also, Usenet participants gave helpful advice.

* Address all correspondence to Ben Shneiderman, Department of Computer Science, University of Maryland, College Park, MD 20742.
7.1 The syntax database
7.2 The parser
7.3 Filters
   7.3.1 Finding best command
   7.3.2 Finding correct arguments
   7.3.3 Example interaction with spsh
7.4 Implementation

PART III. EVALUATION AND CONCLUSIONS

8. Usability evaluation
   8.1 Subjects
   8.2 Logging facilities
   8.3 Administration
      8.3.1 Installation
      8.3.2 Follow-up
   8.4 Problems
   8.5 Statistics

9. Results
   9.1 Interviews
   9.2 Log file

10. General conclusions

11. Suggestions for further research

12. Approach to designing a spelling corrector

Bibliography

Appendix A: The complete command database
PART I. INTRODUCTION

People use computers as tools. Seldom do people use computers for the sake of using computers. They have a task to do. The way that users accomplish their tasks should be as convenient as possible, while still being efficient in terms of machine resources. Typing commands into a computer is powerful and efficient, yet usually requires a good deal of knowledge on the user's part. Furthermore, the computer requires users to type precise commands. For instance, when mistyping a command such as 'catt' instead of 'cat' the computer produces an error message and the user is distracted from the task at hand, so as to cope with the computer interface. Many commentators have suggested that the computer might correct these simple user-computer miscommunications to minimize the disruption.

1. Spelling correction

Much has been written on the design of spelling correctors for word processing. These designs have some relationship to command correction. The design of any spelling corrector must consider extracting the tokens to be checked, the structure of the dictionary of legal words, how to measure the closeness of incorrectly spelled tokens, and the filtering of legal tokens.

1.1 Spelling checking vs. spelling correction

Spelling checking is simply checking if there is an error in spelling. In its simplest form, spelling checking is just looking through a dictionary to see if a given spelling exists. Spelling correction goes beyond checking and attempts to correct the misspelling.

1.2 Extracting tokens

The first step in spelling checking is looking through the input and finding the tokens to be checked. Intuitively, the token is the word. Terminators must be paid particular attention (Peterson 1980).

1.3 Structure of the dictionary

The dictionary is a list of words, which is often stored in the computer with an array of character strings. Other methods of storing the dictionary have been proposed to speed lookup and conserve storage space (Peterson 1980).

1.4 Synonyms and abbreviations

Expert users often use synonyms and abbreviations in place of full commands. The synonyms and abbreviations of the user must be maintained in the spelling interface. If the spelling interface is to correct misspelled synonyms or abbreviations special considerations must be made. For instance, if a user's synonym for some command 'ls -alu' is 'ls' and there is also a command 'as', then which should be chosen when 'ts' is typed?
1.5 Filters

A common way of discussing the correction of misspellings uses two stages: candidate generation and filtering (Dobbing and Cooke 1985). In other words, possibilities are first generated, then some of the candidates are filtered out to form a final list of suggested corrections. However, the candidate generation can be viewed as filtering through all possibilities. The distinction between candidate generation and filtering becomes less helpful when more than one level of filtering is done, especially when filtering in one system becomes the candidate generation of another. Descriptions below will only be discussed in terms of filters.

1.5.1 Methods of testing filters

There are basically two methods of checking the effectiveness of filters: hypothetical methods and implementation tests. The hypothetical tests were the first to be used. These tests take two forms. The first form takes an accepted list of spelling errors and their corrections and tests the percentage of spelling errors which the filter would make. Another method looks through a large amount of real test data and determines the number of spelling errors which could be corrected. This method requires the researcher to make decisions about what the proper spelling is. Implementations tests are more interesting. These tests also take two forms: visible and invisible. The invisible test takes the input and logs the corrections that it would make without letting the user know anything about the possible correction. The most interesting and useful test is the visible implementation test. Here, the corrections are actually suggested to the user.

1.5.2 Simple methods

No filter. One simple solution is simply to have no filter. This means that either 1) the user will have to choose between many possibilities or 2) the words are sufficiently distant so that there is rarely more than one candidate. The no filter solution is often used.

Arbitrary selection. Any arbitrary member of the candidate list is always picked for any given misspelled word. This may be the last or first in the list. This method offers consistency if the same mistake is typed twice. This also means that if the correction is wrong, it will be frustrating to the user.

1.5.3 Exhaustive search

There are essentially two competing considerations in designing an exhaustive search: speed of search and appropriateness of candidates.
Damerau's method. The widely used algorithm by Damerau (1964) matches words with at most one error, one of
1. a wrong letter
2. a missing letter
3. an extra letter
4. a single transposition
This algorithm accounts for 80% of spelling errors due to human errors and machine malfunctions (Damerau 1964), and 80% of spelling errors in the Fortran and CUPL languages (Morgan 1970). Damerau's results are based on word lists checked by computer against a dictionary. Morgan's results are based on his analysis of student's Fortran and CUPL programs. Note that in real systems this percentage will be lower because the dictionary may not have the required dictionary word.

Disagreement counting. The basic idea behind this method is to step through each dictionary entry and the misspelling and count the number of disagreements between them. This method was invented independently by Hobbs and Tribble (1987) and DWIM ("DWIM" 1983), both described later. This method, which can account for multiple errors, is a more liberal method than Damerau's.

1.5.4 Computational Methods
Computational methods attempt to compute a set of candidates instead of searching through the dictionary. These methods have not been used in any of the systems discussed in this paper, but are methods to consider for future systems. Morgan proposed this general method in 1970. Mor and Fraenkel (1982) designed a hashing system for Damerau's method. Other methods include the Soundex system (described in Dobing and Cooke 1985) and the digram and trigram method (described in Peterson 1980).

1.5.5 Context checking
If there is a formal syntax for the input to the corrector, then the type of each word can be considered. For instance, if 'cm main.c' is typed in UNIX, 'cd' is filtered out because the command cd takes a directory as an argument. This method will be considered in detail in the UNIX command syntax section.

1.5.6 Usage trends
There are two types of usage trends: static and dynamic. The static usage trends are those that describe the relatively stable general users. The dynamic usage trends deal with the changing individual user.

Static Usage Trends. Static usage trends deal with trends that do not change from login to login of the user. The basic idea is that trends that users make in a study are used to make corrections for all users. For instance, if the command 'cd' is typed 25% of the time in some given operating system shell, then
the misspelled word 'cm' is probably 'cd' and not something else. There are two types of static usage trends: 1) local system usage trends and 2) overall usage trends.

Damerau's method was implemented in the C programming language by Kernighan and Pike (1984). The implementation also added a measure of the degree of misspelling according to which of the four types of errors occurred. Pollok and Zamora (1984) found that the degree of misspelling was backwards in the Kernighan and Pike implementation. Pollok and Zamora found that degree of misspelling was in the following order: omission >= transposition > insertion >> substitution.

Dobing and Cooke (1985) extensively study static usage trends. They found static usage trends ("error pattern filters") to be "most effective." They classify five types of errors when entering text:

1. common spelling mistakes
2. visual confusion
3. motor errors
4. hand-eye coordination
5. subconscious editing

Dynamic Usage Trends. Dynamic usage trends deal with trends which the user makes. As Peterson (1980) points out, consistency of what is typed is a possible way of correcting misspellings. In a word processing example, if 'griy' is spelled, then 'gray' should be provided instead of 'grey' if the rest of the words are spelled in American English. In a UNIX example, if a user spends a session working on a C program and types 'cm', the C compiler, 'cc', may be more likely than 'cd' (change directory), which would be the correction made by static usage trends.

1.6 Programs with spelling correction

Several types of programs have spelling correction added to them. The most popular is word processing, which has difficulties with context checking when dealing with any natural language. Operating system shells have many advantages over word processors for using spelling correction. Compilers and batch systems are other areas of possible spelling correction.

Word processors. Although word processors are by far the most common application using spelling checkers and correctors, they have several inherent problems. These are 1) large size of dictionary and 2) lack of formal syntax of English sentences. Most of the literature discusses word processing applications. See Peterson (1980) for an overall discussion of several techniques.

Operating system shells. Since operating system shells have a formal syntax, context sensitive spelling correction can be accomplished. The extendability of the dictionary must be kept
in mind. There are two types of possible extensions: user extensions and system-wide extensions. Example shells will be considered in a later section.

**Batch systems.** Morgan (1970) is concerned with spelling correction in compilers. He has had success in a spelling correction CUPL compiler used by students. Mosteller (1983, 1988) did an invisible study on a spool control batch system. Mosteller found that the "vast majority" of errors were conceptual, not typographical (1983). He proposes that installation of spelling correction would be simple in other systems (1988).

2. User considerations of command languages

There are several considerations in the design of command languages (Shneiderman 1980 1987, Schneider 1984), but only those directly relating to the design of a spelling corrector are discussed here.

2.1 Errors versus slips

Lewis and Norman (1987) make a distinction between errors and slips. An error occurs when the user intentionally types something into the computer that is incorrect. A slip is when the user intends to type something other than what is typed.

2.2 Types of users

When designing a command language, it is important to consider the broad range of skills of the users (Shneiderman 1987, Schneider 1984). Using Shneiderman's model of user knowledge, the novice has little syntactic or semantic computer knowledge. "Many forms of slips do not occur with beginners" (Lewis and Norman 1987). The experts have thorough syntactic and semantic knowledge. They "demand rapid response times, brief and less feedback . . ." (Shneiderman 1987). The "automated behavior of the expert leads to the lack of focused attention that increases the likelihood of some forms of slips" (Lewis and Norman 1987). The casual user's main difficulty is remembering syntax.

2.3 Interruption

It is important to consider whether to interrupt the user. Interruption may be disorienting (Field & Spence 1988). "Users often perceive system-initiated help as an interruption" (Kearsley 1988). The design of the interruption must be such that the interruption only interrupts the typing of commands temporarily and not the task at hand. For instance, it may take longer to initiate the help or find the problem than to be immediately reminded.
2.4 Pros and cons of spelling correctors

While many designers have advocated spelling correctors there are many potential disadvantages.

Possible advantages of spelling correctors are:

INCREASES EFFICIENCY. The user does not have to retype commands or take the time to correct them.

PRESERVES TRAIN OF THOUGHT. The user does not have to lose the task train of thought.

DECREASES FRUSTRATION. There is no feeling that small, unimportant details are interfering with the task at hand.

Possible disadvantages of spelling correctors are:

REDUCES CONTROL. The user may feel out of control of the situation.

FEAR OF MISSES. An incorrect correction may be more jarring than smoothing out the train of thought.

ENCOURAGES SLOPPY WORK HABITS. The user will type sloppier.

In Hobbs and Tribble's (1987) study, the novices with the spelling corrector made many more typographical errors (20 times more, on the average).

DECREASES SPEED OF MACHINE EXECUTION. Extra response time to check spelling may be required.

DISTRACTION. If mistakes are not made frequently enough, the user may be jarred every time the spelling corrector makes a suggestion.

3. Human Factors of UNIX

Typically, the human factors aspects of UNIX are considered very poor. The advantages of UNIX and patterns of UNIX command use will also be discussed.

3.1 Advantages of UNIX

The advantages of UNIX are described well in (Kernighan and Mashley 1981). They describe the functional simplicity of UNIX: files, simple programming interface, i/o redirection, program connection, pipes, and shell programming.

3.2 Disadvantages of UNIX

The disadvantages of UNIX are discussed in Aaronson (1982) and Norman (1981). The major disadvantages according to Aaronson are 1) syntax is inconsistent, 2) feedback is sparse, and 3) the "nonmnemonic command names are hard to learn and easy to forget."

3.3 UNIX command use

The major study of UNIX command use is by Hanson, et al. (1984). They found that the top 20 commands were used 70% of the
time. Also, they found that commands are typed in predictable clusters. The errors, which accounted for 10 percent of all commands, were characterized in three types: inconsistent syntax, unclear status, and planning without feedback.

4. Existing Shells

A few shells have been implemented which include spelling correction. These all do some degree of context checking. It is important to consider the benefits and advantages of past shells, correlated with empirical results of studies if available.

4.1 RdMail Interface

Durham, Lamb, and Saxe conducted a study on the RdMail interface, an electronic mail interface (1983). They used context filter, Damerau's method, and then no filter. This means that for each word in the command, only words of the correct type are considered. Then, among words of the correct type they used Damerau's method and all the candidates found here were offered by the system. Keywords are corrected, where each command is a sequence of keywords and parameters. Here is an example interaction:

headers meetngs intersect (since "Jan l") intersect 50:175 <cr>
% Do you mean 'Meetings' instead of 'meetngs'? [No]: y <cr>

This system corrected 27 percent of all erroneous keys over 145,972 commands actually used. The researchers suggested that spelling correction was easy to implement in RdMail and would be simple for compilers and operating systems.

4.2 DWIM

The DWIM (Do What I Mean) interface for Interlisp on the Xerox Star attempts to correct simple mistakes of programming such as spelling mistakes (Teitelman 1981, "DWIM" 1983). This is the most complicated correction system discussed in this section. The filter uses a disagreement counting approach, where closeness is related to percentage of match. DWIM also uses dynamic usage trends. Spelling correction is ordinarily invoked on unbound atoms and undefined functions. Depending on the type of what is mistyped (an editor or function for example), the spelling list checked is different. DWIM is very user-extendible and the user has control over how it operates. Here is an example of interaction with DWIM ("DWIM" 1983):

*DEFINEQ ((FACT (LAMBDA (N) (COND ((ZEROP N 1) ((T (ITIMS N (FACCT 8SUB1 N) (FACT)
*FACT(3]
N9 [IN FACT] -> N ) ? YES
[ IN FACT] (COND -- ((T --)) ) ->
(COND -- (T --))
ITIMS [IN FACT] -> ITIMS
FACT [IN FACT] -> FACT
8SUB1 [IN FACT] -> ( SUB1 ? YES
6
*

4.3 UNIX shells

4.3.1 csh

Csh (C - syntax shell) is specifically a terse shell. Unix is known for its short, abbreviated commands. No spelling correction is built into csh. Csh contains builtin functions of its own. Shell scripts and programs can be written by the user to form new commands.

4.3.2 Hobbs & Tribble csh

Hobbs & Tribble (1987) conducted a study on novice users using a very small (3 command) subset of csh. Only 3 commands were used because the subjects were total novices. Furthermore, they only had 20 minutes to complete as many tasks (such as looking through files for certain words) as possible; this short time was used to induce errors.

The correction algorithm used context filtering, the disagreement counting method, and arbitrary filter for filtering, always generating only one finalist if any. The exact disagreement counting algorithm used only considered matching tail and head of tokens. Here is a sample interaction (user's input underlined):

```plaintext
--- > ls <cr>
Spelling error: Is the command ls? <cr>
Corrected line: ls

apple
grapefruit
orange

--- > cat grapefruit <cr>
Spelling error: Is the command cat? <cr>
Spelling error: Is the first argument grapefruit? <cr>
Corrected line: cat grapefruit

The grapefruit is a sour fruit.

--- > dir grapefruit <cr>
No such command.
Try another command.
```

It was found that the users without the spelling corrector liked the system more than those with the spelling corrector (see Table 1). This can be explained by the nature of novices. The novice has minimal syntactic knowledge. Asking whether the
Table 1

<table>
<thead>
<tr>
<th></th>
<th>With Spelling</th>
<th>Without Spelling</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subjective Satisfaction Mean</td>
<td>( \mu = 4.806 )</td>
<td>( \mu = 5.472 )</td>
<td>( T = 1.743 )</td>
</tr>
<tr>
<td></td>
<td>std. = 0.827</td>
<td>std. = 0.795</td>
<td>df = 16, p &lt; 0.1</td>
</tr>
<tr>
<td>Syntactic Errors Rate</td>
<td>( \mu = 1.88 )</td>
<td>( \mu = 0.096 )</td>
<td>( T = 2.313 )</td>
</tr>
<tr>
<td></td>
<td>std. = 0.098</td>
<td>std. = 0.067</td>
<td>df = 16, p &lt; 0.034</td>
</tr>
<tr>
<td>Task (Semantic) Errors Rate</td>
<td>( \mu = 0.939 )</td>
<td>( \mu = 0.946 )</td>
<td>( T = 0.087 )</td>
</tr>
<tr>
<td></td>
<td>std. = 0.182</td>
<td>std. = 0.139</td>
<td>df = 16, p &lt; 0.932</td>
</tr>
<tr>
<td>Task Performance</td>
<td>( \mu = 0.422 )</td>
<td>( \mu = 0.547 )</td>
<td>( T = 1.298 )</td>
</tr>
<tr>
<td></td>
<td>std. = 207</td>
<td>std. = 0.201</td>
<td>df = 16, p &lt; 0.213</td>
</tr>
<tr>
<td>Efficiency</td>
<td>( \mu = 0.768 )</td>
<td>( \mu = 0.860 )</td>
<td>( T = 0.817 )</td>
</tr>
<tr>
<td></td>
<td>std. = 0.242</td>
<td>std. = 0.235</td>
<td>df = 16, p &lt; 0.426</td>
</tr>
<tr>
<td>Mean Backspaces Per Line</td>
<td>( \mu = 0.767 )</td>
<td>( \mu = 0.482 )</td>
<td>( T = 1.401 )</td>
</tr>
<tr>
<td></td>
<td>std. = 0.444</td>
<td>std. = 0.419</td>
<td>df = 16, p &lt; 0.180</td>
</tr>
</tbody>
</table>

Subjective satisfaction is measured by answers on a questionnaire (higher score means the more they liked the system). Syntactic error rate was the number of suggested spelling corrections versus the total number of commands. Task error rate was measured by the number of correctly answered questions over the total number of questions answered. Task performance was the amount of correctly answered questions compared to the total questions. Efficiency is number of commands used divided by number of correctly answered questions. Backspaces per line was simply the number of backspaces used per line.
"first argument" should be something else is disorienting for the user. Also, the term "error" is negative, which Shneiderman (1987) warns against. Why not just delete the "Spelling error:" prefix? Or better yet, why not just ask whether a whole command is correct, like the following:

```bash
---
ct graphfruit <cr>
Did you mean "cat grapefruit"? <cr>
---
```

The grapefruit is a sour fruit.

```bash
Notice that two lines of output are eliminated. Lewis and Norman (1987) point out that the view of error messages should be that of an apology, not an accusation of error. For instance, a still better response might be:

Sorry, your command, as typed, cannot be parsed, did you mean "cat grapefruit"? <cr>

Experts will need a different type of message, one that is fast and explicit. For instance, a good message may just be the brief corrected form:

cat grapefruit? <cr>
```

4.3.3 tcsh

Tcsh is a modified csh, with several added features such as limited spelling correction and line editing commands. Tcsh uses limited context checking, Pollok and Zamaru's ambiguity resolution, and finally, an arbitrary filter. To tcsh, a command is structured as follows:

```bash
<cmd> <files or directories>
```

Here is a sample interaction (were the file main.c is to be compiled with the 'cc' compiler and the executable to be named 'out'):

```bash
% cm -o out main.c <cr>
cm: Command not found.
% cm -o out main.c
  ^ use editing commands to get last line, and move cursor here. Now, if hit <esc-s>,
% c -o out main.c
  ^ it got a command but not the desired one
% cc -o out main.c
  ^ if add c here and hit <cr>,
No source file main.c
% cc -o out main.c
  ^ if get last command and hit <esc-s>,
% cc -o out main.c
  ^ now hitting <cr>,
```
4.4 Analyzing these shells

These shells can be analyzed in terms of benefits and disadvantages, summarized in the following table. RdMail and the Hobbs and Tribble shells will be analyzed using statistics when they were collected from the studies. Where statistics are not available, either personal experience or speculation from its operation are used.

<table>
<thead>
<tr>
<th></th>
<th>RdMail</th>
<th>H&amp;T</th>
<th>tcsh</th>
<th>DWIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>efficiency</td>
<td>?</td>
<td>no effect</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>train of thought</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>frustration</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>control</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>misses</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>sloppy</td>
<td>?</td>
<td>5</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>speed</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>mistake frequency</td>
<td>3</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

For Benefits:
1 = excellent
2 = good
3 = ok
4 = fair
5 = poor

For Disadvantages:
1 = no problem

Table 2. Comparison of spelling shells
PART II. DESIGN AND IMPLEMENTATION

The most difficult aspect of the study was the design and implementation. Design decisions are discussed, and then comments on how the implementation, named spsh, may differ from the original design.

5. Interfacing with Unix

5.1 When to Invoke the spelling checker

Deciding when to invoke the spelling checker is crucial. If it is always invoked, then time is wasted. If it is not called when needed, the interface will be inconsistent.

An obvious time is to check the command is when it "fails." Common examples are "out of memory," "no such file name," "no such device," "process killed," and "too many files open." Note that some of these suggest checking spelling ("no such file name," "no such device") but the others do not. Furthermore, consider commands which may have arguments misspelled yet have not way of figuring whether they are spelled correctly. For instance, when calling the editor emacs with the command "emacs new," where new is a new filename, should it be checked?

Unfortunately, the return codes from the commands are not standard. The general pattern is the commands return 0 when they do not fail, and any other value when they do fail. The problem is that some commands return 0 when they fail and that some commands return non-zeros when they do not fail.

5.2 How to Hook In

There were basically three paths available to hooking the spelling algorithms to the shell: 1) Calling the shell with each command, 2) Set up the shell through a pipe, and 3) modify csh code. The problem with 1) is speed because the shell has to be loaded, then the entire initialization would take place for every command. Method 1) was used in Hobbs and Tribble (1987) because the customized shells which experts use are not necessary with novices. The problem with 2) is that we cannot check return codes as discussed above, and it would be difficult to even tell when a command was done. Option 3 was used, simply because the others were not acceptable. The reason 3 (which is obviously the most powerful) was not chosen immediately it that it is by far the most difficult to implement. Note that a "parsing on the fly" method (Seidel 1986) was not considered because the command line should only be parsed by the spelling checker when it needs to (the line will already be parsed by the shell and command when it is correct).
command line should be:
checked    not checked
----------------------------------
failed : corrected ; lost time ;
----------------------------------
succeeded : uncorrected ; saved time ;
----------------------------------

Table 3. Summary of problems with failure checking

command line should be:
checked    not checked
----------------------------------
non-zero : corrected ; lost time ;
----------------------------------
zero : uncorrected ; saved time ;
----------------------------------

Table 4. Summary of problems with return code checking

6. Unix Command Syntax

6.1 Different Types of Command Syntax

6.1.1 Commands which can be described with simple BNF

First, consider simple commands such as

---- % ls
----  main.c  main.h  bin/

This command simply lists the contents (files and directories) in the current directory. Consider a command which takes only one argument.

---- % cat main.h

/* Include file for main.c */

Note that here, only a file is acceptable. Consider a command with flags:

---- % cc -o out main.c

Here, the -o flag signifies that the next argument is an output file where the executable will be placed. Note that main.c is an input C source file.

These commands can be described by the following general syntax:
<command invocation> ::= <flag bundles> <regular arguments>
In specific,
<cc invocation> ::= <cc flag bundles> <cc regular arguments>
<cc regular arguments> ::= <files>
<cc flag bundles> ::= <cc flag bundle> <cc flag bundles> ; <>
<cc flag bundle> ::= -o <file> ; -I<directory> . . .

6.1.2 YACC

YACC is a parsing facility for programmers to use when programs require parsing which can be described in BNF. The basic form for YACC grammar definition can be seen in a translation of the above BNF for the cc command.

%token STRING
cc_invocation: cc_flag_bundles cc_regular_arguments
cc_regular_arguments: files
cc_flag_bundles: cc_flag_bundle cc_flag_bundles
cc_flag_bundle: '-o' file
| '-I' directory
|
file: STRING
directory: STRING

To make this grammar work, a lexical analyzer must be written. The STRING token is returned when a string is read. This analyzer reads strings separated by non-alphabetic characters, and '-' whenever the type is a flag (preceded by a '-').

To make YACC useful, it should do something depending on how the invocation is parsed. Adding an action to each rule is necessary. For instance, a portion of YACC code might be:

file: STRING { $$ = file_spell($1); }
directory: STRING { $$ = directory_spell($1); }

This means that whenever we get to file: and directory: we return ($$) as the correct spelling of the file or directory of the first component (STRING, otherwise known as $1).

6.1.3 Commands which Cannot be Described With BNF

Consider a command of the following form:
---- % kermit clb /dev/dial06 2400

This telecommunications command means connect (c) to the server, setting up the line (l) as /dev/dial06 and as baud rate
(b) at 2400. The basic problem is that BNF does not support the semantics of the form \(<\text{flag}_1> \ <\text{flag}_2> \ <\text{arg}_1> \ <\text{arg}_2>\), that is, the types associated with \(<\text{flag}_1>\) must match \(<\text{arg}_1>\). Consider how this command must be parsed:

1) Read the command,
2) Read first character,
3) Check if it is a flag,
4a) If so, read next character,
4b) If not, is it actually a device name?
5) Is it a flag?
6) Yes, and it takes a file (devices are files) as argument. Be ready to read a file (can't be anything else).
7) Read next flag, it is takes a number as an argument. Be ready to read a number (can't be anything else).
8) Read next character, is it a flag? If no, read the rest of the word without losing the character just read. See what the next argument should be. Is it an argument of a flag or a regular argument? It is a file, because back at step six the name was remembered.
9) Read last argument. . . .

6.1.4 Commands to be supported

There are several reasons for wanting to support all commands, but the fundamental one is consistency. However, if all commands are supported, are they all supported equally. It may not be worth adding all commands for two reasons 1) speed of lookup and 2) difficulty of parsing.

6.2 Argument Types to be Supported

A more critical decision than which commands to support is what argument types to support. Argument types define whole classes of tokens which will or will not be supported:

1. Files
2. Directories
3. Files or directories
4. Login names
5. Machine names
6. Command names
7. Strings
8. Numbers
9. Flags

Criteria for support argument types include:

FREQUENCY. How often is the argument type used. If it is not used much, it may not be worth supporting. If it is used in most commands, then it should certainly be supported.

RANGE. What are the possible values? If there are too many possible values, then it may be impossible to make a good guess. Also, it may take too long to search all possibilities.

COMPUTABILITY. Is the range computable?
UNIQUENESS. How unique is each member in the range? In other words, how easy is it to use a simple candidate generator and get a single candidate?

The criteria for selecting the argument types are summarized in the following table. The implementation only supports directories, filenames, and filenames or directories.

<table>
<thead>
<tr>
<th></th>
<th>FREQ</th>
<th>RANGE</th>
<th>COMP</th>
<th>UNIQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Files</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Directories</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Files or directories</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Login names</td>
<td>4</td>
<td>2</td>
<td>?</td>
<td>4</td>
</tr>
<tr>
<td>Machine names</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Command names</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Strings</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Numbers</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Flags</td>
<td>1</td>
<td>?</td>
<td>?</td>
<td>5</td>
</tr>
</tbody>
</table>

Frequency: 1 - very frequent <--> 5 - very infrequent
Range: 1 - very narrow <--> 5 - very wide
Computability: 1 - easily comp. <--> 5 - difficult to compute
Uniqueness: 1 - each very unique <--> 5 - range very alike

TABLE 5. Summary of Argument Types

6.3 Syntax Parameterization
Since context filtering was elected, a parser had to be built. It seems clear that YACC cannot be used after seeing the kermit command example. Even if commands that could not be described in BNF were not supported, there is a significant problem with using YACC. The problem occurs when a new command is to be added to the spelling corrector. There are 2 problems: 1) every administrator of a spelling shell cannot be expected to know YACC, and 2) too much space would be used for the YACC-produced code. The solution is one that takes the designer longer, yet saves space and effort later: syntax parameterization. It seems that a small vector of values can represent all existing and (reasonably) possible syntaxes of UNIX. After carefully consulting the UNIX section 1 manual these basic parameters were chosen:
1) command name
2) command flags,
   where each flag is parameterized by
   a. flag name
   b. delimiter type (+ or -)
   c. zero or one arguments
d. type of argument
e. does the argument follow immediately (without a space in between)

3) regular argument type
4) are dashes required before flags
5) should numbers or strings be skipped
6) number of arguments

7. The syntax database, parser, and filters

7.1 The syntax database
For simplicity, the entire database was read into memory during the shell execution. There are two reasons for this: 1) it is easier to do, and 2) since an exhaustive search is done, it is necessary to traverse the whole list of commands frequently. After this decision is made, it becomes important to make the syntax parameterization representation as small as possible for each command. An efficient method of sorting these commands is also necessary.
In memory, for each command the structure contains
1) a null-terminated character string
   => representing the command name
2) 2 bytes for the information of general information on the command: 4 bits for the type, 3 bits for number of regular arguments, 1 bit for whether dash required for flags, and 2 bits for type to skip.
3) 1 byte for each flag: 2 bits for number of arguments, 1 bit for storing whether argument is immediate, 1 bit for storing if delimiter is '+' instead of '-', and 4 bits for type.

These command information structures are simply kept in a linked list because they must be searched linearly anyway.

7.2 The parser
The parser has several interesting features:
1) a dequeue structure
2) recallable from errors
3) a backtracking feature

A dequeue structure is a structure which can access the last element inserted and the first element inserted. The reason for this will become clear when parsing the following example:

cc -o out -I/usr/include main.c

First, the -o is read and o takes an other type. The next word is not a flag, so the first type inserted in the dequeue structure should be the type of out. Since the first type inserted was other, out is acceptable. Get next flag, -I, and put the type it takes into dequeue structure. This takes an
immediate argument. So, get the last type put in the dequeue structure. Check as a directory. It passes. The next word does not start with a '-' (cc requires them) so it must be an argument. What type is the argument? Get the first type put in the dequeue structure. There is none, so all flags have satisfied argument requirements. This means that main.c is a regular argument, which is a filename for cc. It passes. The whole line passes.

What if there is an error when parsing? The parser exits (saving relevant information and returning current flag and argument information) and lets the calling routine deal with the error. Consider a short example:

cd dirr

Assuming the only close directory name is 'dir', the parser get the arguments to the command cd. It recognizes that dirr is not a flag, so it gets the first element in the dequeue structure, which does not exist. So it checks this as a regular argument of cd, of type directory. It fails. So the parser returns to its caller information about the type the argument should be and the argument position. Assuming that somehow the 'dirr' is corrected to 'dir', and parse is called again, it will continue parsing where the error was before.

7.3 Filtering
This filter narrows the range of possibilities before Damerau's algorithm gets hold of the command words.

7.3.1 Finding best command
To find the best command requires calls to Damerau's algorithm during three phases. First, the possible commands are listed with Damerau's algorithm. Next, the parser is called for each command. The error recovery property is necessary in finding the best command. When an error is found by the parser, control is returned to the routine finding the best command. Here Damerau's algorithm is called again to see if a close candidate exists. If not, then the candidate is not considered. If so, then the parser is started up again. The routine finding the best command counts the number of times the parser fails. The command which causes the parser to fail the fewest times is picked. This command's arguments are then changed as a normal guessing argument procedure.

7.3.2 Finding correct arguments
The parser is started up. When it fails, control is back at the guessing routine. Using Damerau's method, an attempt is made to fix the word. If it cannot, the correction fails. Otherwise, the parser is started up again.
7.3.3 Example interaction with spsh

If the example given in the tcsh section were run through spsh it would produce this session:

% cm -o out main..c
cm: Command not found.
The command has been corrected
? cc -o out main.c
y<cr>
%

Here, spsh is much faster for the user than tcsh at correcting this error for these reasons: 1) The suggestion was correct. 2) It all happened automatically. 3) A complete command was offered.

7.4 Implementation

The complete database in this study had 87 commands. The choice of commands was based on personal experience and user requests (see user evaluation section). The file which contained the database was only 1707 bytes, with an average of 19.6 bytes per command. The parser did not implement the skipping of arguments.
PART III. EVALUATION AND CONCLUSIONS

The shell was given to 21 UNIX users and a log file was maintained on the users. After the users had used the shell for about two weeks, they were interviewed.

8.1 Subjects
The 21 subjects were faculty, graduate students, undergraduates, and staff using one of five UNIX systems at the University of Maryland running csh version 5.3. The users were either people that the researcher knew or users responding to a message sent over electronic mail.

8.2 Logging facilities
A facility was added to the shell which would keep the following information:
1. Number of commands typed at each session.
2. Line of each command which had non-zero return code with either:
   a) Correction guess and whether accepted.
   b) Reason spelling corrector could not make correction
The results of the log files were compiled with a shell script.

8.3 Administration
8.3.1 Installation
The installation was in three steps:
0. If the shell was not installed on the system at all yet, the shell was put onto the system.
   1. Put spsh as last command in .login [start spsh at each login]
   2. Put .cmd.data in home directory [syntax data must be in home directory for startup]
   3. Put 'set ignoreeof' in .cshrc [forces users to use the exit command to exit shell]

The users were told that if they wanted any commands to be added, they could request having the commands added to the syntax database.

8.3.2 Follow-up
Questions
Several questions were intended to be posed to the users of spsh:
1. Did you like the shell?
2. Would you like to keep the shell?
3. Would you keep the shell with modifications?
4. How fast do you type?
5. How well do you know UNIX?
6. What do you use the system primarily for?

Procedure of follow-up
0. Ask questions
   1. get .cmd.log and .cmd.data from their home directory
   2. Take spsh invocation out of .login

8.4 Problems
The major implementation problem was a disk crash on the
system where most of the subjects were on. This made two
problems: premature interruption of spsh, and often the loss of
of the log file. After this and other problems, 15 acceptable
users were left: 11 users with data files, and 14 user's with
interviews.

9. Results
The users used the system from 25 days to 2 days (one user
quickly took out the shell himself), and averaged about two
weeks.

9.1 Interviews
The most interesting questions were the first three. Five
out of fourteen users did not like the shell, 2 users "did not
know" if they liked it (because corrections were proposed so
rarely), 3 thought the shell was "ok" or "so-so", and 4 liked the
shell. Seven of the users kept the shell, and the other seven
did not keep the shell. There were no strong feelings in favor
of the shell, but a few strong feelings against the shell.
People were too unpredictable on the question about whether they
would keep the shell with modifications. For instance, someone
replied they "didn't know" when he just said he would keep the
shell the way it was. The other questions were not used in any
statistics, because they were vague, as was pointed out by
several users. The basic problems can be summarized in the
following categories:

1. Bad idea Some people thought the whole idea was bad,
citing that they either wanted to correct it themselves or it did
not help enough to matter.
2. Inconvenient logout Since the shell is not yet an
official shell, spsh was run as a subshell. This meant that two
exit commands had to be typed to logoff the system.
3. Annoying messages The most annoying message was the
"Command not in database" message. This message was meant to
prompt the user to send mail requesting this command to be added
to the database.
4. Lost features Some features were lost in spsh. One
feature was a feature in the 'cd' command, which will search directories outside of the current directory if necessary. Since spsh just looks at the current directory and cd is a builtin (so the corrector is always invoked), these uncommon 'cd' commands will fail. One user usually used tcsh, which has many more features than csh.

9.2 Log file

After collecting the 11 log files, it was discovered that not all logons and logouts were registered. Since the logouts and logons were not all registered, other messages may not have been written to the log file. The first one or two logons which were obviously the installation process or the user testing out the system were deleted. At least 5,662 commands were typed in the shell. Almost half of these commands were typed by one user.

10.6 percent of commands were checked, meaning the command was either a builtin, failed, or was not a command at all. Spsh proposed a correction for 0.6 percent of the commands. Of these proposals, 72.2 percent were accepted. The "Command not in database" error occurred in 7.1 percent of all commands. This was due largely to the evolving database: often users who got this error received a new, expanded database. In the log file, errors of this type suddenly disappear. Different users may have a different number of commands in the database.

10. General Conclusions

Several conclusions can be made. One of the most important conclusions is actually a question: is spelling correction worth the effort? The spelling corrector in this study offered corrections on only 0.6% of all commands typed in. Furthermore, users were not enthusiastic about spsh. However, most were enthusiastic when installing it. After the amount of effort already put into the developing system, it may be worth remedying the relatively small amount of the problems found (compared to the number of problems already overcome). Three more specific conclusions can be made:

1. Designing spelling correction systems is not necessarily simple. Many researchers have suggested that adding spelling correction to systems is simple (including Mosteller 1988, and Durham, et al. 1983). There are two reasons why the design and implementation of past batch systems and interactive shells has been relatively simple: centralized parsing and well-behaved symbol table lookup, which is the case in DWIM, Morgan's compiler, RdMail, and Mosteller's system. In csh, parsing and symbol table lookup is done differently for each command.

Another way of making the design simple is to only partially describe the syntax. This method was taken by tcsh. Spsh also used this method, to a lesser extent (still, not all commands'
syntax can be described by the syntax parameters).

2. Minimize distraction from user's task. When adding a correction facility it must not distract from the task at hand. The "Command Not in Database" error message is an example, discussed below. Another example of distraction from user's task is typing logout twice, instead of simply typing the accustomed one command. This points to a more subtle type of distraction when a feature a user is used to is no longer available. The lost features discussed in the results section show cases where the attitude toward the implementation can be greatly affected by deviance from the familiar environment.

3. No error message may be better than a specific error message. Critics of UNIX have cited UNIX's sparse error messages as a problem (Norman 1981). However, the sparse error messages may lead to less distraction from the task at hand. Mosteller (1983) found that more specific error messages improved his batch system. Shneiderman (1987) suggests having specific error messages. In general, a specific error message makes sense: the user has a better idea of how to remedy the situation. More variables should be considered: what was the error message before the more specific one, how often will the message occur, and what are the consequences of a less precise (or nonexistent) error message.

The "Command not in database" was intended to prompt the user to ask for the command to be added to the database. Consider the three variables suggested above. First, there was no error message generated before when the command was not in the database. Second, the message occurred frequently (7.1% of all commands). Third, if the message was taken out, the only thing that would happen is these commands would not be corrected. When designing the message, consistency was of primary concern (either support all commands or not at all). Now, it seems that a loss of consistency (some commands not being added to the database) may have been better than an annoying error message.

11. Suggestions for further research
Several area of research are suggested:

If the various problems noted by users were fixed, would users like the system?
Would an addition like optional spelling corrections without the user's confirmation make experts happier?
Reconsideration of the syntax parameterization and developing better parameters might make the system better. For instance, commands like kermit and tar could be defined in terms of positional syntax.
Methods of automatically adding new commands to the syntax
database could be considered.

Research is necessary on whether single-choice corrections (more likely to be wrong) are preferred by users over multiple-choice corrections (requiring the user to stop and choose).

Throughout this paper, it is assumed that context filtering is the most appropriate method for operating system shells. This is justified by the fact that 1) the shell's commands must have a formal syntax, and 2) user's may get upset when a suggested correction is obviously of the wrong type. Why not consider other filters without context checking, especially when parameterizing the syntax is difficult?

12. Approach to designing a spelling corrector

The first step in designing a spelling corrector for an operating system shell is to determine whether the effort of designing the corrector will be worth it. These considerations should be made (each feature is progressively more difficult to implement):

1. Is parsing done in one place? If so, then adding a spelling corrector should be simple.

2. If parsing is done in several places, do each of the parsing points use the same table lookup routines? If so, then adding a spelling corrector should be relatively simple.

3. Is the syntax of commands describable in BNF? Is there a system parser which describes BNF (like YACC)? (If not, do as in 5.) If so, then decide when the corrector should be invoked. If the spelling corrector is always to be invoked, then consider "parsing on the fly".

4. Is most of the syntax of commands describable in BNF? It may be worth leaving out the commands which are not (as discussed in commands to be supported section). If so, then do the same as in 3.

5. Can the syntax be described by some other form? Is some parser available on the system which can parse this? If so, then do same as 3. Otherwise, seriously consider whether the project is worth it.

If the spelling correction project is worth it, then remember these hints:

1. What types of arguments will be corrected (as discussed in argument types to be supported section)?

2. Decide on what types of filters to use (it is already assumed that context checking is done). If code is available for a filter, use it. Consider spelling correcting abbreviations and synonyms.

3. Change as little original code as possible.

4. Decide on the control the user will have over the system.

5. Try to anticipate the types of users that will be using
the system. Consider running a log of commands that current users type.

6. Throughout the design, talk with potential users.
BIBLIOGRAPHY


