Towards a Helpful Shell: Scenarios and Commands

Saumya Ray

Abstract

Programmers can face many challenges when using the command line for programming and other tasks. These include problematic runtime errors, accidental file deletion, tracking the cause of an introduced error in a previously working program, forgetting commands, concepts, and previous work context, and not being able to estimate the time spent on or required for a task. SuperShell is our extension of the Bash command line that can be helpful in these scenarios. Help is provided through several new features that provide (a) detailed command history in logs, (b) aggregate usage information through periodic emails and commands, (c) suggestions for runtime errors through a hierarchy of predefined and programmer-defined rules, and (d) recovery of past versions of a file. SuperShell can be embellished with new features such as group statistics, new programmer-defined rules, and adaptation of suggestions based on responses to previous suggestions.

1 Introduction

Learning how to use a shell for the first time can be daunting and unfamiliar to beginner programmers. It can be challenging to learn how to properly use Bash command syntax, pipelining, and I/O redirection [1]. New learners often start programming on visually interactive interfaces and IDEs, a stark contrast to the shell. Error messages shown in the shell by non-interactive compilers and at runtime may not be very helpful. Students who typically learn Java or Python as their first programming language using an IDE can find the shift to programming with C in the command line to be different and difficult.

With this in mind, our goal was to create a shell that enhances the user experience and knowledge gained. The subgoals include making suggestions

to users when an error occurs; providing shell-based versioning; integrating shell history with versioning implicitly; recording detailed command history with full command, associated file, its contents, and standard input, standard output, standard error, and timestamp; searching the detailed command history; locating which version introduced an error that exists in the latest version; helping users go back to a previous version of file due to erroneous edits, deletes, and overwriting; giving students feedback to encourage good work ethic through insights about past work through periodic programming summary; and providing context for breaks or interruptions in work.

We have developed a tool called SuperShell with these features. In Section 2 we illustrate the new SuperShell features through use cases. In Section 3 we define the new commands and their implementation. In Section 4 we list suggestions for future work. In Section 5 we summarize our work.

2 Use Cases

The following hypothetical examples show how Alice and Bob, two friends taking the UNC introductory C programming course, COMP 211, use SuperShell. Alice and Bob are new to programming in C and are using a shell for the first time in this course. Alice likes to start assignments early while Bob starts a few days before the deadline. Both are prone to making typing errors. Alice and Bob are familiar with the features of SuperShell and are eager to use these features effectively.

2.1 Unable to resolve runtime error → shelp

Alice is writing her first C program, `ex1.c`. After editing, saving, and closing the file, she tries to run the file and sees the following output.

```
$ pico ex1.c
$ ./ex1
./ex1: No such file or directory
Enter 'shelp' for any suggestions regarding error output
```

Figure 1: Error when running `ex1.c`

Used to implicit compiling by an IDE, Alice did not compile the C file. As a result, she gets an error message "`./ex1: No such file or directory`." This message confuses Alice because she was certain the file

she had edited (`ex1.c`) existed. It is not very useful in helping Alice figure out what went wrong.

SuperShell has a feature called `shelp`. If enabled, when an error is detected, SuperShell will provide suggestions that may resolve the problem. Alice sees the message "`Enter 'shelp' for any suggestions regarding error output`" below the first error message as she did not have `shelp` enabled. She enters the command and sees the following response.

```
$ shelp
SuperShellHelp
Auto Helping...
1 Make sure you have compiled ex1.c
To compile, enter the following command:
gcc ex1.c -o ex1
Do any of these numbered suggestions look relevant?
If so, enter the number (e.g. 1) or simply press enter: 1
Try suggestion #1
$ gcc ex1.c -o ex1
$./ex1
Hello, World!
```

Figure 2: Using `shelp` command to get suggestion(s) for error in Fig. 1

Alice sees SuperShell's numbered suggestion(s). The first suggestion asks her to make sure that she has compiled `ex1.c`, and shows her how to compile it using gcc. Then `shelp` asks Alice whether any suggestion listed was relevant and to enter the number of the suggestion. Now Alice has an idea of what is causing her error. She enters the suggested command `gcc ex1.c -o ex1`, then executes the file `ex1.c`, and sees the expected output "Hello World." SuperShell was able to clarify the initial vague error message and provided a relevant solution to Alice's problem.

If appropriate rules are defined, `shelp` can be used to identify the cause of an error and provide a solution.

2.2 Accidental File Deletion → sundo

Bob is working on a program that will add all input numbers together. He is exploring different ways to implement this and has multiple files (`add.c`,

add1.c, and add2.c) for this. Bob decides that he only wants to use the first file (add.c) and enters a command to delete the other files (add1.c and add2.c). The command entered was `rm add*` which removed the executable add, and all three source files add.c. Bob incorrectly assumed that the `rm add*` command would remove only the files with numbers after the string "add." He sees his error when attempting to run add, as he gets the error message "`./add: No such file or directory. Enter 'shelp' for any suggestions regarding error output.`"

```
$ pico add.c
$ pico add1.c
$ pico add2.c
$ gcc add.c -o add
$ rm add*
$ ./add
./add: No such file or directory
Enter 'shelp' for any suggestions regarding error output
```

Figure 3: Unable to run a file that has been erroneously deleted

Bob is not sure why he is seeing this error, so he enters the `shelp` command. SuperShell help displays two suggestions for this error. The first suggestion is "`Make sure you have compiled add1.c. To compile, enter the following command: gcc add.c -o add.`" The second suggestion is "`Check to make sure this file exists. If accidentally deleted, try sundo command: sundo -f add.c.`" Then, as before, `shelp` asks, "`Do any of these suggestions look relevant?`" and to enter the number of the relevant suggestion. Bob knows that he definitely compiled `add.c`, so the first suggestion is not useful for him. However, the second suggestion hints that Bob may have deleted the files incorrectly. Bob enters suggestion number 2 before issuing the command `$sundo -f add.c`.

```
$ shelp
SuperShell Help
Auto Helping...
1 Make sure you have compiled add.c
To compile, enter the following command:
gcc add.c -o add
2 Check to make sure this file exists
If accidentally deleted, try sundo command:
```

```
sundo -f add.c
Do any of these suggestions look relevant? If so, enter the
number or press enter 2
Try suggestion # 2
```

Figure 4: Using `shelp` command to get suggestions for error in Fig. 3

The `sundo` command is a SuperShell command. It shows the last edited version of a given file. Using the sundo command with the file `add.c`, Bob can get the most recent version.

```
$ sundo -f add.c
"#include <stdio.h>
int main() {

 int a = 1;
 int b = 3;
 int c = a + b;
 printf(\"%d + %d = %d
\", a, b, c);
 return 0;
}"
```

Figure 5: Using `sundo` command to see the last saved version of `add.c`

If he wishes to, Bob can see the previous `n` versions of `add.c` by specifying a number flag in the sundo command. To see the last three versions of the file, Bob can enter `sundo -f add.c -n 3` in the command line.

```
$ sundo -f add.c -n 3
"#include <stdio.h>
int main() {
  return 0;
}"
"#include <stdio.h>
int main() {
 int a = 1;
 int b = 3;
 int c = a + b;
  return 0;
}”
```

```
"#include <stdio.h>
int main() {
 int a = 1;
 int b = 3;
 int c = a + b;
 printf(\"%d + %d = %d
\", a, b, c);
 return 0;
}"
```

Figure 6: Using `sundo` command to see the last three saved versions of `add.c`

Using the sundo command, Bob copies the saved text of `add.c` in the clipboard. When he opens `add.c` using a text editor, Bob sees that this file is empty. He infers that this file got deleted in his previous `rm` command because the code he had written was gone. He pastes the copied text and saves his changes. Now Bob can run `add.c` after compiling it. He realized that he used the `rm` command incorrectly and resolved to understand more about the command. The SuperShell command `sundo` helped Bob recover and restore a previously deleted file.

The `sundo` command can be used to see past versions of a file so users like Bob can restore the file to a previous state.

2.3 Encountering a new error in a previously working program → shistory

Alice works on a simple program that determines if an integer input is even or odd using a switch statement. In her file, `switch.c`, she designates a variable `x` to store the value of the integer the user enters. In her switch statement, Alice writes a case for an even input. In the even case, the program prints "`x is even`," in which `x` is the integer input.

```
#include <stdio.h>
int main()
{
   int x;
   scanf("%d", &x);
   switch(x%2){
```

```
      case 0:
         printf("%d is even", x);
         break;
   }
   return 0;
}
```

Figure 7: `switch.c` file with only the even case

She tests her program with an even number input (4) to ensure her program works as expected.

```
$ ./switch
4
4 is even
```

Figure 8: Running `switch.c` with input "4" and correct output message

Then Alice adds another case for odd number inputs and a default case. Each case prints out a statement saying whether or not the integer input is even or odd. In the default case, the statement is "`Will this ever output?`"

```
#include <stdio.h>

int main()
{
   int x;
   scanf("%d", &x);

   switch(x%2){
        case 0:
           printf("%d is even", x);
        case 1:
           printf("%d is odd", x);
        default:
           printf("will this ever print?");
           break;
   }

   return 0;
}
```

Figure 9: `switch.c` file with all cases

Alice compiles and runs this program again. She enters an even number (4) again, but now she sees that all print statements are showing.

```
$ ./switch
4
4 is even
4 is odd
will this ever print?
```

Figure 10: Running `switch.c` with input "4" with incorrect output messages

Alice isn't sure why all statements are printing, especially since entering an even number before worked as expected previously. Alice wants to confirm that her program was working as expected before and inspect the differences between the two files.

To do so, she decides to search the history log to see the history of commands with `switch.c`. SuperShell has a special command history that records various attributes of an entered command (time, command, associated file, standard output, standard error, and more). Alice uses the file and command flags in the `shistory` command to narrow her search. First she searches the history with the command "`./switch`." This will only show history logs for when Alice ran the switch program. Each record will show the standard output for the command, so Alice verifies that her program was initially working correctly when she had just the even case.

```
$ shistory -c ./switch
{
  "time": "2020-05-03:10:39",
  "command": "./switch",
 "full_command": "./switch",
  "stdout": "4 is even",
  "stderr": "",
  "filename": "",
  "file": ""
}
{
  "time": "2020-05-03:10:40",
  "command": "./switch",
```

```
  "full_command": "./switch",
  "stdout": ""4 is even\n4 is odd\nwill this ever
   print?"",
  "stderr": "",
  "filename": "",
  "file": ""
}
```

Figure 11: Using `shistory` command to verify that `switch.c` previously ran correctly

Next, Alice uses another `shistory` command to view the edit history of the file switch.c to compare the file contents. Since she has edited `switch.c` multiple times, if Alice simply searches `shistory` on the file `switch.c`, she may see too many edit logs. This would not be helpful in quickly identifying what changed. Therefore, Alice narrows her search in `shistory` to display logs for `switch.c` with the command flag set as `gcc`. This will show file contents for all compiled versions of `switch.c`.

```
$ shistory -f switch.c -c gcc
{
  "time": "2020-05-03:10:40",
  "command": "gcc",
 "full_command": "gcc switch.c -o switch",
  "stdout": "",
  "stderr": "",
  "filename": "switch.c",
  "file": "#include <stdio.h>\n\nint main()\n{\n int x;\n
scanf(\"%d\", &x);\n \n switch(x%2){\n case 0:\n printf(\"%d is
even\", x);\n// case 1:\n// printf(\"%d is odd\", x);\n//
default:\n// printf(\"will this ever print?\");\n }\n \n return
0;\n}"
}
{
  "time": "2020-05-03:11:34",
  "command": "gcc",
 "full_command": "gcc switch.c -o switch",
  "stdout": "",
  "stderr": "",
```

```
  "filename": "switch.c",
  "file": "#include <stdio.h>\n\nint main()\n{\n int x;\n
scanf(\"%d\", &x);\n \n switch(x%2){\n case 0:\n printf(\"%d is
even\n\", x); break;\n }\n return 0;\n}"
}
```

Figure 12: Using `shistory` command to compare compiled file versions

Looking at the file contents in each log displayed from this command, Alice notices that the previously compiled version had a break statement after the print statement in the even case. That break got moved to the default case. Alice realizes that for her program to work correctly, she needed a break statement at the end of each case for her program to work as expected.

The `shistory` command can verify that a program was previously working and can be used to compare logged file contents and output to pinpoint the cause of an error

2.4 Ascertaining relative assignment difficulty → mailstats

Alice and Bob are talking about their classes in the past week. Bob thinks that week was tough because of COMP 211, while Alice thought 211 was easier last week. They want to see who was right. Alice and Bob remembered that they usually get an email every week about their SuperShell usage that summarizes how much time they spent coding, how many files they edited, a list of unique commands they used, and how many lines and words were added to files. A sample SuperShell email is shown below. Bob and Alice see that they both spent less than thirty minutes coding in shell last week, so they conclude that COMP 211 was not the cause of Bob's tough week.

```
SUBJECT: Your SuperShell Usage

========= ::: SuperShell Weekly Usage ::: =========
>  Total time: 23 minutes 53 seconds
>  Lines of code: 59
>  Word count: 180
```

```
**************************************************
9 files edited:
     1  ".bash_profile"
     2  "ex1.c"
     3  "hello.c"
     4  "switch.c"
     5  "mail.txt"
     6  "offline.sh"
     7  "add.c"
     8  "repeat.sh"
     9  "text.c"

17 unique commands:
     1  "$HOME"
     2  "./ex1"
     3  "./hello"
     4  "./switch"
     5  "./text"
     6  "bash"
     7  "cd"
     8  "chmod"
     9  "export"
    10  "gcc"
    11  "ls"
    12  "pico"
    13  "printenv"
    14  "python"
    15  "r"
    16  "rm"
    17  "setenv"
```

Figure 13: Example of weekly email sent to user

The SuperShell email is useful in alerting and informing users about their coding habits for the past week. They do not need to run a command to see their usage summary, as sending emails is automatically configured in SuperShell.

Inspired by analogous emails from fitness and grammar checking applications, SuperShell sends weekly emails that show users statistics about

their usage the previous week, including time taken, word/line count, files used, and unique commands.

2.5 Estimating assignment completion time → sstats

Alice and Bob have an assignment due in two days. Alice has already done the homework assignment but Bob has not yet started. Bob wants to estimate how long this will take him, so he asks Alice how much time she spent coding the assignment. Alice knows she didn't spend too much time working on the assignment, but doesn't know exactly how long it took. Checking her weekly SuperShell email will not help, because she wants information on a single file. Instead, she uses the `sstats` command in SuperShell. In the command she enters, Alice indicates which file (named `lab3.c`) she wants to see statistics on.

```
$ sstats -f lab3.c
Your stats for file lab3.c:
Total time: 10 minutes 17 seconds
Lines of code: 10
Word count: 35
```

Figure 14: Using `sstats` command to see statistics for `lab3.c` file

With a file specified, the `sstats` command displays the total time, lines of code edited, and word count of the file. Alice can now share with Bob that she took under eleven minutes to complete the assignment.

The `sstats` command can be used on a single file to see how long a user spent working on it.

2.6 Determining pre-break context → sstats

Bob is in COMP 211 and other computer science courses. Spring Break was two weeks ago and he wants to know the context of what he was specifically working on in COMP 211 the week before break. To refresh his memory, Bob can use the `sstats` command specifying a start date to show statistics from. Since March 1, 2020 was the first day of the week before Spring Break, Bob can use this date in the command. This will show statistics based on all SuperShell usage since the specified date. However, to narrow his search and ignore irrelevant information, Bob can use a more detailed command and specify a directory to get the specific context for COMP 211. The command

`sstats -d Documents/COMP211 -s 2020-03-01` will show information starting from March 1 in the directory Documents/COMP211. Seeing a list of files he was working on and commands he was using can help Bob remember what was going on in his class after a long break.

```
$ sstats -d Documents/COMP211 -s 2020-03-01
Your stats starting from  2020-03-01 for directory
Documents/COMP211:
Total time: 7 minutes 46 seconds
Lines of code: 208
Word count: 3627
8 files edited:
"add.c"
"bubble.c"
"courseinfo.txt"
"ex2.c"
"hello.c"
"hpfiller.txt"
"lab5.c"
"lab6.c"

5 unique commands
"grep"
"rm"
"pico"
"gcc"
"bash"
```

Figure 15: Using `sstats` command on a given directory `Documents/COMP211` from a start date March 1, 2020

The `sstats` command can be used on a specific directory starting from a given date to show what files have been used since that time.

2.7 Reacquainting with pre-break workflow → sstats

Alice is starting a new assignment after Spring Break but has forgotten how to use a shell for the edit, compile, and run workflow. Alice cannot search the history to see how she used specific commands since she does not remember them. `sstats` can show her how many unique commands were

used in a time period. Alice specifies that the time frame is a week by using the -w week flag.

```
$ sstats -w
Your stats starting from  2020-04-01:
Total time: 46 minutes 24 seconds
Lines of code: 1025
Word count: 42005

14 files edited:
"./ex2.c"
"add.c"
"add1.c"
"bubble.c"
"deletethis.txt"
"disable_help.sh"
"ex1.c"
"ex2.c"
"hello.c"
"hpfiller.txt"
"long.txt"
"num.txt"
"repeat.sh"
"repeated.sh"
"textchanges.c"

45 unique commands (listing 10)
"./hello"
"./lab3"
"sed"
"ls"
"grep"
"help"
"vim"
"gcc"
"pico"
"grep"
```

Figure 16: Using `sstats` command to see statistics from the past week

Her memory of the previous commands can be used to filter out from the returned list the commands she commonly used in the edit-compile-run workflow.

The `sstats` command can be used to see what commands have been used within a past week, month, or year

2.8 Error in command previously used correctly → shistory

Alice has used input/output redirection previously. She wants to redirect the output of her `hello.c` program to the file `helloout`. However, she incorrectly types the wrong arrow in the command `./hello < helloout`, and gets a generic error message "`helloout: No such file or directory`."

```
$ ./hello < helloout
helloout: No such file or directory
```

Figure 17: Error in attempt to use I/O redirection

Since Alice remembers using output redirection correctly before on the file `hello.c`, she can search the history logs to find the full correct command. In this command, Alice uses the SuperShell command `shistory` with the -c command flag to find examples of when she used I/O redirection when running the file `hello.c`. The `shistory` command logs information about each entered command including `full_command`.

```
$ shistory -c ./hello
{
"time": "2020-03-30:11:47",
"command": "./hello",
"full_command": "./hello > helloout",
"stdout": "",
"stderr": "",
"filename": "",
"file": ""
}
```

Figure 18: Using `shistory` command on a past command

After using the `shistory` command and seeing how she was able to use output redirection without any errors, Alice can use redirection properly in her current task.

The `shistory` command can be filtered by past commands entered to show users examples of how they previously used the command correctly

2.9 Reviewing for a quiz or exam → shistory

Bob is studying for an upcoming Linux commands quiz. He wants to review how he used commands such as `grep` to process large text files. He wants to see the different ways he used the `grep` command, so he enters the `shistory` command and specifies that `grep` is the command he is searching for. The output for `shistory` includes the full command entered and its standard output. Bob can see all the ways he used various options of the `grep` command and see its outputs.

```
$ shistory -c grep
{
"time": "2020-04-20:11:48",
"command": "grep",
"full_command": "grep \"lavender\" hpfiller.txt",
"stdout": "Boggarts lavender robes, Hermione Granger Fantastic
Beasts
and Where to Find Them. Bee in your bonnet Hand of Glory elder
wand,
spectacles House Cup Bertie Botts Every Flavor Beans
Impedimenta.
Stunning spells tap-dancing spider.",
"stderr": "",
"filename": "",
"file": ""
}{
"time": "2020-04-20:11:50",
"command": "grep",
"full_command": "grep \"*giant\" hpfiller.txt",
"stdout": "",
"stderr": "",
"filename": "",
```

```
"file": ""
}{
"time": "2020-04-20:11:51",
"command": "grep",
"full_command": "grep -i \"Nearly\" hpfiller.txt",
"stdout": "Half-giant jinxes peg-leg gillywater
broken glasses large
black dog Great Hall. Nearly-Headless Nick now string them
together,
and answer me this, which creature would you be unwilling to
kiss?
Poltergeist sticking charm, troll umbrella stand flying cars
golden
locket Lily Potter.",
"stderr": "",
"filename": "",
"file": ""
}{
"time": "2020-04-20:11:51",
"command": "grep",
"full_command": "grep -c \"harry\" hpfiller.txt",
"stdout": "0",
"stderr": "",
"filename": "",
"file": ""
}{
"time": "2020-04-20:11:52",
"command": "grep",
"full_command": "grep -cv \"Hermione\" hpfiller.txt",
"stdout": "7",
"stderr": "",
"filename": "",
"file": ""
}
```


Figure 19: Using `shistory` command on a past command "`grep`" to review usage

Searching the history logs with a specific command helps Bob review how he has used the command with different parameters in the past.

The `shistory` command can be filtered by specific commands to show full commands and their outputs and errors

2.10 Using new interactive commands → add_interactive.sh

Alice has heard of the next text editor `heels` and wants to use this in SuperShell. To do so, she needs to designate `heels` as an interactive command. Interactive commands must be designated as such to correctly record: a) lines of text (written code or words) in a file, (b) time spent editing a file, and (c) a version of the file before each interactive session.

The command `$ bash add_interactive.sh heels` adds the command given (`heels`) to a list of interactive commands. Then, when SuperShell is executing commands, it can properly record history. Current interactive commands SuperShell supports include `pico`, `nano`, `vim`, `jupyter`, etc.

Any interactive command needs to be designated as such in SuperShell by using the `add_interactive.sh` script

2.11 Learning more about SuperShell functionality → sinfo and usage feedback

Each SuperShell command described in the previous scenarios has multiple flags that customize and filter the output. Alice and Bob are only aware of a few flags they can use with commands like `shistory` and `sstats`, so they enter the following commands to become aware of all the flags they can use in their SuperShell.

```
$ sinfo
SuperShell is a tool to help programmers using a shell.

Special commands are

* sinfo

* shistory
Useage: shistory [-f filename] [-c command] [-d start_date]
start_date format: YYYY[-MM[-DD]]
```

```
* sstats
Useage: sstats [ -w | -m | -y] [-f filename | -d
directory_name] [-s start_date]
start_date format: YYYY[-MM[-DD]]

* sundo
Useage: sundo [-f filename] [-n number of versions]

* shelp
Shows numbered suggestions after an error is detected

SuperShell help can be enabled or disabled by entering
enablehelp or disablehelp.

Interactive commands can be added by entering
bash add_interactive.sh <new command>
```

Figure 20: The `sinfo` command shows how SuperShell features can be used

If they use a flag incorrectly or try to use a flag that does not exist, Alice and Bob will automatically see the information explaining the flags.

```
$ shistory *
0 is not a recognized flag!
Usage: shistory [-f filename] [-c command] [-d start_date]
start_date format: YYYY[-MM[-DD]]
$ sstats *
0 is not a recognized flag!
Usage: sstats [ -w | -m | -y] [-f filename | -d directory_name]
[-s start_date]
start_date format: YYYY[-MM[-DD]]
$ sundo *
0 is not a recognized flag!
Usage: sundo [-f filename] [-n number of versions] [-d
start_date]
start_date format: YYYY[-MM[-DD]]
```

Figure 21: Entering "`*`" after a SuperShell command shows its usage

The `sinfo` command shows users information about the features available in SuperShell

3 Special Commands and Implementations

3.1 sinfo

The `sinfo` command lists all special (new) commands available in SuperShell with their various options (Figure 20). It also shows how to change additional settings, such as adding interactive commands, enabling `shelp`, and changing the welcome message. This command works by displaying the text of a file called `supershellinfo.txt`, which contains this information.

3.2 shistory

The `shistory` command allows users to view their detailed command history stored in JSON format. By default, SuperShell tracks every command issued and stores the command in a history file. Instead of just using up and down arrow keys to navigate past commands entered, or using the `history` command to see a list of full commands entered, the `shistory` command outputs a more comprehensive history. This history allows for filtering based on date, command, and filename. The history for each day is stored in its own JSON file. All JSON history files are located in a `SuperShellHistory` directory.

After a command is entered and executed, SuperShell records the history details of the command. Useful metadata, including the time the command was issued, the full command, the standard output of the command, and any standard errors generated are stored (Figure 11, 12, 18, 19). Furthermore, if the command has an associated file that is edited, properties such as lines of code written, words written, time spent working (in minutes and seconds), and a copy of the file contents is stored. All of this information together forms an entry that is stored in the history JSON file for the current day that is named `SuperShellHistory-[YYYY-MM-DD].json`.

To show the history of shell usage, a user enters the `shistory` command with optional flag parameters. The -c flag filters history by the command

given, the -f flag by the file name given, and the -d flag by the date given (in YYYY-MM-DD format). The parameter information given is used to create a `jq` query. `jq` is a command-line JSON processor. Based on the given date range, output logs from files stored for each date that matches the query. The default time range uses the entire history log.

3.3 sstats

The `sstats` command uses the data stored in `shistory` and gives the user the capability to see statistics about a file or directory within a specified time period (week, month, or year). It shows time spent coding, the number of lines and words changed, how many files were edited, and five selected unique commands (Figures 14, 15, 16). By default, this command shows stats of commands for the current week.

This command works by first parsing the arguments passed in the flags. The -y, -m, -w, -d flags limit the statistics shown to the past, year, month, week, or from the starting date, respectively. The flags are used to determine the beginning of the timeline for the statistics. Based on the earliest date within this time frame, all the relevant history JSON files are combined. The information from the flags is used to create a `jq` query. Different forms of this query are to get the minutes, seconds spent, lines of code, word count, a list of files edited, and unique commands used. These statistics are then echoed to the shell screen.

3.4 shelp

The `shelp` (SuperShell help) command suggests reasons for errors and possible solutions to resolve the detected errors. When standard error is detected, SuperShell checks various “Super Shell Rules” to determine the help message and suggestions. To provide some of the information these rules may use, it sets the following environment variables before calling them: The suggested help uses the values of the following environment variables: `LAST_MODIFIED`, `LAST_COMPILED`, `LAST_EXECUTED`, `LAST_FILENAME`, `LAST_STDERR`, `LAST_STDOUT`, and `LAST_COMMAND`. Except for the last three variables in this list, all of these variables refer to file names. All environment variables are set and updated when history is being recorded.

`shelp` can be enabled or disabled by entering `enablehelp` or `disablehelp`. By default, it is disabled. If it is not initially enabled, and the user gets an error, they can get assistance by just entering `shelp` in the shell.

After the command is executed, if any standard error is detected, then either the `shelp` command automatically runs or a message "`Enter 'shelp' for any suggestions regarding error output`" appears.

What `shelp` actually does is run all rule scripts in a given rule directory. The default rule directory is `SuperShellRules`. The rule scripts are named `rule-<general name>.sh` and are located in the rule directory. The rule scripts contain many if-statements, and if any of them are true, the associated suggestions are stored in a temporary suggestions file. The if-statements check if the environment variables meet certain conditions.

In Figure 22, the if-statement first checks if the file name in the environment variable `LAST_FILENAME` does not exist. If this is the case, then there are several reasons why there will be an error message. This particular rule applies to the example in Section 2.1, in which a user tries to execute a file that has not been compiled. The executable for the file does not exist, so SuperShell help suggests to compile the file.

```
if [ ! -f "$LAST_FILENAME" ]; then
     echo "Make sure you have compiled $LAST_EXECUTED" >>
$HOME/.tmpsuggestions.txt
     echo "To compile, enter the following command:" >>
$HOME/.tmpsuggestions.txt
     echo "gcc $LAST_EXECUTED -o $LAST_FILENAME" >>
$HOME/.tmpsuggestions.txt
fi
```

Figure 22: Example of if-statement in rule script

If there are multiple subdirectories within the given rule directory, `shelp` will go through rule scripts within those subdirectories. After all suggestions are gathered in a file, they are numbered and then displayed to the user. Each suggestion provides a possible reason for the error and a potentially helpful command that can help the user resolve the error. As we saw in the examples, the user is asked to read through the suggestions and enter the

number of a relevant suggestion; they can enter the suggested command if they choose to do so.

The order of suggestions displayed is based on the numbering in Figure 23. First the rule scripts in the rule directory are executed. After that, the rule scripts in subdirectories are executed.

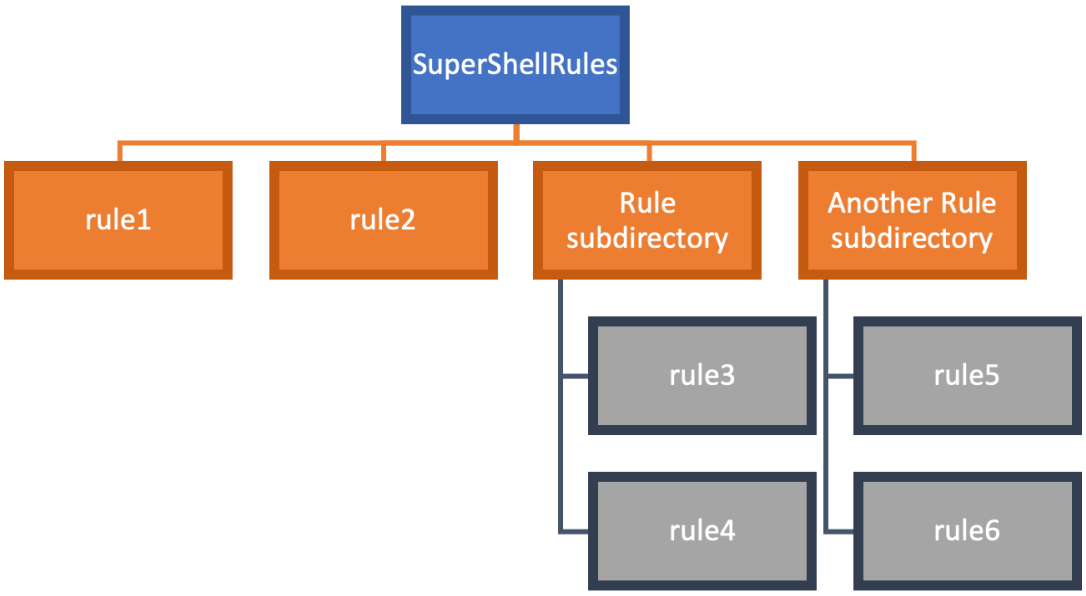

Figure 23: Hierarchical structure of the rule directory

As another example, SuperShellRules can be used to suggest corrections to misspelled commands. If a user entered "`cd Donloads`" and `shelp` was enabled, they would immediately get a suggestion that they may have misspelled "`Downloads`" and a suggestion to create a directory called "`Donloads`" since it does not exist.

```
$ cd Donloads
cd: Donloads: No such file or directory
SuperShell Help
Auto Helping...
1 Make sure you have created the directory you are navigating
to!
Suggestion:
mkdir Donloads
2 Did you mean “Downloads?”
Suggestion:
cd Downloads
Do any of these suggestions look relevant? If so, enter the
number or press enter
```

Figure 24: The `shelp` command giving spelling suggestions

3.5 sundo

As described in section 3.2, any time a file is edited in SuperShell, a copy of the file text is stored in the history logs. The `sundo` command can show a number of previous versions of an edited file. This will allow users to recover deleted files or restore a previous version of a file.

This command has two flags: -n and -f. The -f flag indicates which file's contents should be shown; this flag is mandatory. The -n flag allows users to decide how many past versions of a file they want to see. The default value for this is 1, and the maximum number of versions displayed depends on the edit history of the file. The command searches all history logs that include the filename in the file field, gets the copy of the file contents stored, and outputs it to the user.

3.6 mailstats

`mailstats` is an internal command that sends users weekly emails about their SuperShell usage. This is based on the `sstats` command and can be configured with email-specific parameters. The command constructs a formatted email message and sends it to the user if the current day is Monday. Once the email has been sent, an internal flag is set so that SuperShell does not send multiple emails in a week.

3.7 Code Implementation

Using a shell script, SuperShell intercepts user input and the output of executed commands and then interprets the input and output. The SuperShell script runs on top of the user's existing Bash shell. The user's Bash profile is updated to run the SuperShell script. This way, SuperShell runs immediately upon opening a shell window and displays a welcome message. The SuperShell script interprets whether a command entered is a special SuperShell command, an interactive command, or a general command in order to appropriately execute it and log the history. After a command is executed, the SuperShell script displays the output and checks if there are any errors. If so, the `shelp` command will generate suggestions accordingly. Figure 25 shows the flowchart of the SuperShell script.

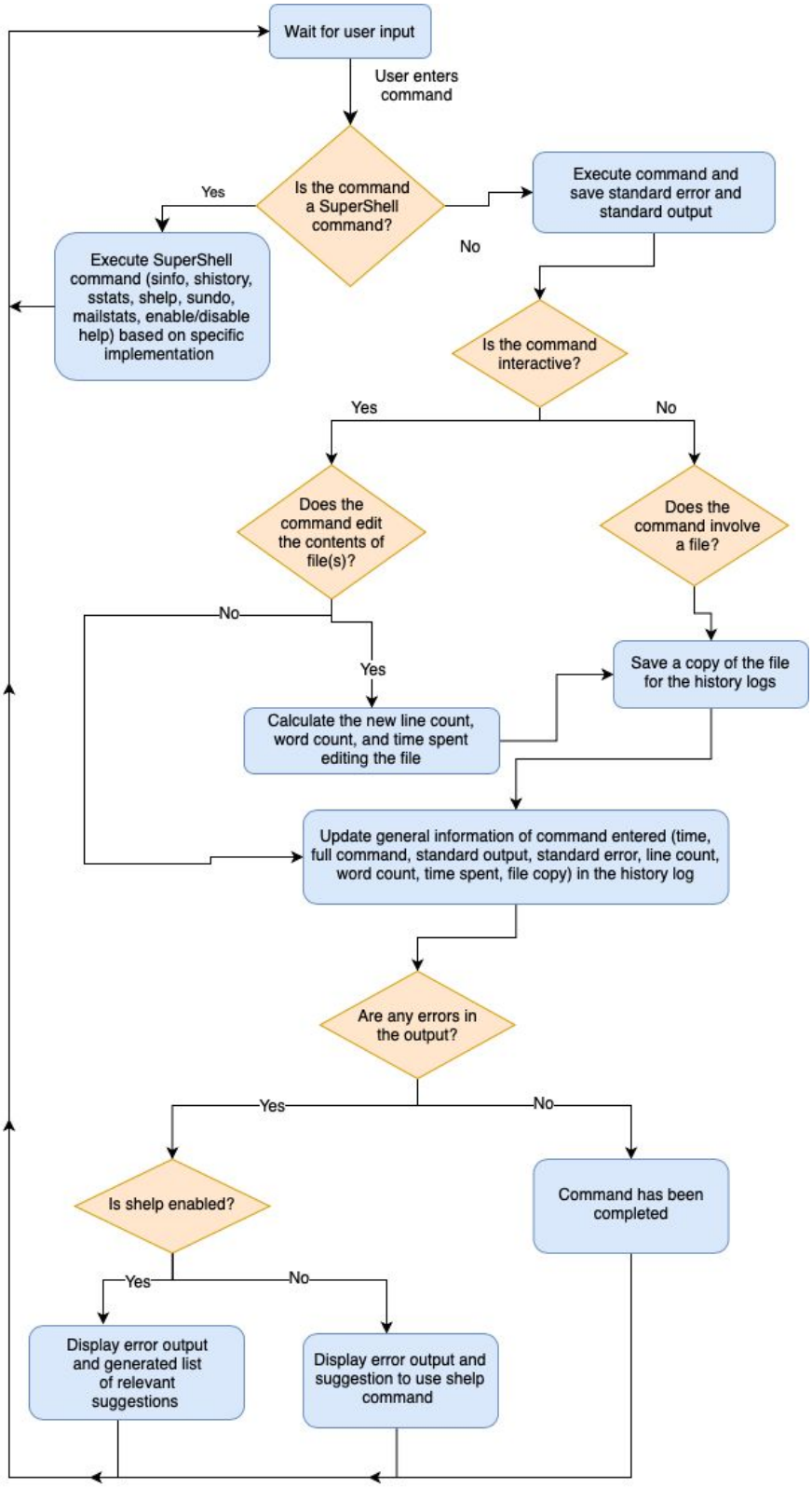

Figure 25: A flowchart of the main SuperShell script

4 Future Work

The current SuperShell can be expanded to add more useful capabilities in providing more support to beginner programmers.

4.1 Group Statistics

Currently, students are able to access information about their personal shell usage in various ways. However, in order to compare their statistics with others, they must ask other students directly. It would be helpful to have a group statistics tool that collected usage data from all students in a class and aggregated them. Such a tool would allow students to compare their performance to the rest of their class. If a student notices that they are taking much longer to complete assignments than others, they may realize that they should go to office hours to get help on the assignments.

4.2 New Programmer-Defined Rules

The current rules defined in SuperShell Rules can be expanded to account for other kinds of errors that may need new environment variables. These rules can be applied to users in a specific course or group. There can be a rule that enforces a naming convention for files. Or, there can be a rule in which a command always has to have standard output. A group can specify which rules they want to use for a group of users.

4.3 Data-Driven Suggestions

SuperShell help lists possible numbered suggestions that may help resolve the latest error. At this time, `shelp` asks the user to enter the number of the suggestion that they think is most useful. This data collected can be used to predict which suggestions are more relevant to particular errors. These can be numbered based on the likelihood of being helpful with the number one suggestion being predicted as the most relevant.

The `shelp` command can also be expanded to use the history of successful users who have recovered from errors without using SuperShell help. It can see what commands they used to resolve errors they faced, and give those suggestions to other users using `shelp`.

5 Summary

In this paper, we discuss how a shell can be developed to be useful to students in a variety of scenarios in which they face issues or want to improve their efficiency.

- Inability to resolve a runtime error
    - If appropriate rules are defined, `shelp` can be used to identify the cause of an error and provide a solution.
- Accidental File Deletion
    - The `sundo` command can be used to see past versions of a file so users can restore the file to a previous state.
- Encountering a new error in a previously working program
    - The `shistory` command can verify that a program was previously working and can be used to compare logged file contents and output to pinpoint the cause of an error
- Ascertaining relative assignment difficulty
    - SuperShell sends weekly emails that show users statistics about their usage the previous week, including time taken, word/line count, files used, and unique commands.
- Estimating assignment completion time
    - The `sstats` command can be used on a single file to see how long a user spent working on it.
- Determining forgotten context
    - The `sstats` command can be used on a specific directory starting from a given date to show what files have been used since that time.
- Reacquainting with pre-break workflow
    - The `sstats` command can be used to see what commands have been used within a past week, month, or year
- Error in command previously used correctly
    - The `shistory` command can be filtered by past commands entered to show users examples of how they previously used the command correctly
- Reviewing for a quiz or exam
    - The `shistory` command can be filtered by specific commands to show full commands and their outputs and errors
- Using new interactive commands

    - Any interactive command needs to be designated as such in SuperShell by using the `add_interactive.sh` script
- Learning more about SuperShell functionality
    - The `sinfo` command shows users information about the features available in SuperShell

The paper also describes several directions for future work. Group statistics can be used to be more informative about the SuperShell usage of multiple people in a group. More SuperShell rules can be defined by programmers to expand the suggestions given upon an error. Suggestions marked as relevant for particular errors in the past can be used to adaptively list the suggestions given to users in order of decreasing relevance.

References

[1] C. Ramey, "*Bash, the Bourne-Again Shell*," in *Proceedings of The Romanian Open Systems Conference & Exhibition* (ROSE 1994), *The Romanian UNIX User's Group* (*GURU*), November 3, 1994 (pp. 3-5).

Appendix

rule-bashcmds.sh

```
#!/usr/bin/bash

if [[ "$LAST_STDERR" == *"command not found"* ]]; then
     if [[ "$LAST_COMMAND" == "ll" ]]; then
          echo "Do you mean ls?" >> $HOME/tmpsuggestions.txt
          echo "" >> $HOME/tmpsuggestions.txt
     fi
fi
```

rule-compiledfile.sh

```
#!/usr/bin/bash

if [ "$LAST_STDERR" != "" ]; then
    if [[ $LAST_STDERR == *"No such file or directory"* ]]; then
          if [ "$LAST_COMMAND" == "cd" ]; then
```

```
                echo "$LAST_FILENAME"
                if [ "$LAST_FILENAME" == "Donloads" ]; then
                     echo "Did you mean Downloads?" >>
$HOME/tmpsuggestions.txt
                     echo "Suggestion: " >>
$HOME/tmpsuggestions.txt
                     echo "$LAST_COMMAND Downloads " >>
$HOME/tmpsuggestions.txt
                else
                     echo "Make sure have created the directory
you are navigating to!" >> $HOME/tmpsuggestions.txt
                     echo "Suggestion: " >>
$HOME/tmpsuggestions.txt
                     echo "mkdir <directory name>" >>
$HOME/tmpsuggestions.txt
                     echo "" >> $HOME/tmpsuggestions.txt
                fi
          fi
          if [ ! -f "$LAST_FILENAME" ]; then #make more specific
                # echo "" >> $HOME/tmpsuggestions.txt
                echo "Make sure you have compiled $LAST_EXECUTED"
>> $HOME/tmpsuggestions.txt
                echo "To compile, enter the following command:"
>> $HOME/tmpsuggestions.txt
                echo "gcc $LAST_EXECUTED -o $LAST_FILENAME" >>
$HOME/tmpsuggestions.txt
                echo "" >> $HOME/tmpsuggestions.txt
          fi
          if [ ! -f "$LAST_EXECUTED" ]; then
                echo "Check to make sure this file exists" >>
$HOME/tmpsuggestions.txt
                echo "If accidentally deleted, try sundo
command:" >> $HOME/tmpsuggestions.txt
                echo "sundo -f $LAST_EXECUTED" >>
$HOME/tmpsuggestions.txt
          fi
    fi
fi
```

rule-sscmds.sh

```
#!/usr/bin/bash

if [[ "$LAST_STDERR" == *"command not found"* ]]; then
     if [[ "$LAST_COMMAND" == "ll" ]]; then
          echo "Do you mean ls?" >> $HOME/tmpsuggestions.txt
          echo "" >> $HOME/tmpsuggestions.txt
     fi
     if [[ "$LAST_COMMAND" == "rhistory" ]]; then
          echo "Do you mean shistory?" >>
$HOME/tmpsuggestions.txt
          echo "" >> $HOME/tmpsuggestions.txt
     fi
     if [[ "$LAST_COMMAND" == "shidtory" ]]; then
          echo "Do you mean shistory?" >>
$HOME/tmpsuggestions.txt
          echo "" >> $HOME/tmpsuggestions.txt
     fi
     if [[ "$LAST_COMMAND" == "rhisotry" ]]; then
          echo "Do you mean shistory?" >>
$HOME/tmpsuggestions.txt
          echo "" >> $HOME/tmpsuggestions.txt
     fi
     if [[ "$LAST_COMMAND" == "rstats" ]]; then
          echo "Do you mean sstats?" >> $HOME/tmpsuggestions.txt
          echo "" >> $HOME/tmpsuggestions.txt
     fi
fi
```