International Journal of People-Oriented Programming
Volume 5 • Issue 1 • January-June 2016

Natural Shell:
An Assistant for End-User Scripting

Xiao Liu, College of Information Sciences and Technology, Pennsylvania State University, University Park, PA, USA
Yufei Jiang, College of Information Sciences and Technology, Pennsylvania State University, University Park, PA, USA
Lawrence Wu, College of Information Sciences and Technology, Pennsylvania State University, University Park, PA, USA
Dinghao Wu, College of Information Sciences and Technology, Pennsylvania State University, University Park, PA, USA

ABSTRACT

Scripting is a widely-used way to automate the execution of tasks. Despite the popularity of scripting, it remains difficult to use for both beginners and experts: because of the cryptic commands for the first group, and incompatible syntaxes across different systems, for the latter group. The authors introduce Natural Shell, an assistant for enabling end-users to generate commands and scripts for various purposes. Natural Shell automatically synthesizes scripts for different shell systems based on natural language descriptions. By interacting with Natural Shell, new users can learn the basics of scripting languages without the obstacles from the incomprehensible syntaxes. On the other hand, the authors' tool frees more advanced users from manuals when they switch shell systems. The authors have developed a prototype system and demonstrate its effectiveness with a benchmark of 50 examples of popular shell commands collected from online forums. In addition, the authors analyzed the usage of Natural Shell in a lab study that involves 10 participants with different scripting skill levels. Natural Shell effectively assists the users to generate commands in assigned syntaxes and greatly streamlines their learning and using experience.

KEYWORDS

Education, End-User Scripting, Natural Language Programming, Shell, WinBat

INTRODUCTION

The command line interface is essential for users and administrators, as it allows them to fully harness and customize the power of operating systems (Robbins & Beebe, 2005). Shell scripting significantly extends the utility of the command line by automating a batch of commands. However, despite the popularity that shell script enjoys, its syntax is cryptic for beginners and it requires quite a bit of practice before they can put down the manual.

Shell scripting is complicated, because of the rigid syntax and the inconsistency between shell systems. Most commands have "options", which significantly change and extend the functions of commands. However, the use of options is excessively succinct and thus, makes the commands incomprehensible for newcomers to a particular shell. Besides, the large amount of options that each command accepts, adds more cognitive load during the learning process. Take the *grep* command as an

DOI: 10.4018/IJPOP.2016010101

Copyright © 2016, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

example, there are 47 options that *grep* accepts including popular ones like *"-i"* for case-insensitive, *"-r"* for recursive search, and *"-w"* for word only. It is difficult for novices to remember more than a few, and for experts to remember all the options. Moreover, this situation compounds when a user has to get accustomed to the syntax of some other shell as she switches to another shell system. If a Linux user wants to delete a directory in MS Windows, she may encounter difficulties, since MS Windows Command Prompt adopts *del* rather than *rm*, as the command to perform delete operations. Other superficial differences include argument position, case-sensitivity, options for commands instead of separate commands, etc. Even within the UNIX-like OS family, there are Ksh, Csh, Bash, and Zsh shells, each of which has its own features. Because of the difficulties engendered by shell scripting, there is a need for a "unified shell".

Previous efforts have focused on whether to set up a reasoning system for UNIX Shells, or how to augment the UNIX shell with more powerful functions. Lee proposed a natural interface for shell scripting that can reason based on pre-defined cases (Lee & Lee, 1995). The reasoning system behind the interface generates scripts for seasoned users, but it overlooks the uninitiated users. Weaver et al. (Weaver & Smith, 2012) presented another UNIX text-processing tool that extends the original shell to support frequent manipulations on text files. However, their tool is platform-sensitive. We believe that a cross-platform system can handle more cases and be particularly useful for those who work across multiple operating systems.

Natural language is recognized as a more versatile method for beginners to work with programming tasks (Ballard & Biermann, 1979; Dijkstra, 1979; Brill, 1992). The way people perceive knowledge with natural language descriptions differs from that using strict and unintuitive scripting commands and their arcane outputs. Comparisons using different dimensions were discussed including simplicity, affordance, memorability, and common results, concluding that natural language has clear advantages.

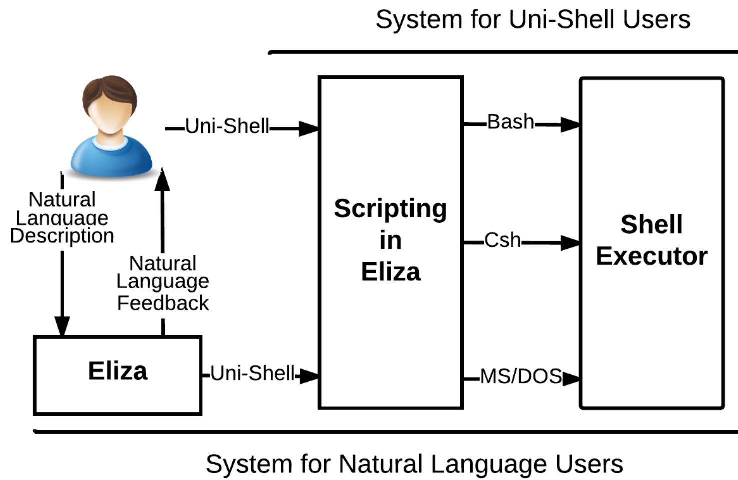
In this paper, we propose a tool that interacts with end-users in natural language and transforms their descriptions into shell scripts, as shown in Table 1. The higher level of abstraction eliminates the syntactical differences between shell systems. Users can write scripts built on bare, pseudo-style logic. As such, we believe that the natural language shell makes shell scripting accessible to typical computer end-users. Our design, which we call Natural Shell is based on Weizenbaum's ELIZA (Weizenbaum, 1966), a chatbot that uses a rule-based method to process English conversations. Natural Shell inherits the rule-based method from ELIZA but adopts regular expressions other than keywords to construct rules. In addition, a generic scripting language is created, called "Uni-Shell", that functions as an intermediary between the natural language interpreter and the target shell commands. Compared to the alternative portages, e.g., Csh or Bash for UNIX and batch for Windows, Uni-Shell is more natural and universal. On the other hand, it is succinct enough such that frequent users may prefer it, compared to taking natural language as the input.

The whole script generation is an interactive process as can be seen in Figure 1. The system consists of three parts: ELIZA, SiE (Scripting in ELIZA) script and Shell Executor. These three processes form a pipeline. ELIZA module first processes natural languages descriptions to synthesize the commands in Uni-Shell and then responds in natural language to confirm. Then, SiE script takes in the Uni-Shell commands from the ELIZA module and synthesizes the target shell commands, which are then picked up and executed by the Shell Executor. Instead of typing with natural languages, commands in Uni-Shell are also provided for users as a second choice. Users can directly input a

Table 1. Natural language descriptions and the associated Shell/Batch commands

Natural Language Descriptions: Remove Directories that Contain "foo" in the Name
Csh: if grep $name "foo" rm -r $name endif **Bash:** if grep $name "foo" rm -r $name fi **Batch:** if FINDSTR /r %name "foo" RD /s/q %name

Figure 1. Natural shell system architecture



complete script in Uni-Shell and our tool will generate a script in any specified syntax for them and then pass it to the Shell Executor module for execution. Results are displayed in response to the execution of the generated command/script.

To the best of our knowledge, Natural Shell is the first shell that uses conversations between users and computers to automatically and adaptively generates scripts in three different syntaxes. The key idea of our work is to utilize the regulation of natural language descriptions and to understand the users' intentions according to the rules we designed. In the rest of this paper, we first introduce related works and the arguments that initiated our idea. We briefly summarize the elements of the user interface we designed and the corresponding functionalities of each panel. Then, we describe our key techniques by going through an example. It demonstrates the whole process that generates a script of multiple shell commands. We elaborate on our contribution in end-user scripting with regard to two end-user problems we frequently sighted in online discussion forums. To demonstrate the effectiveness and analyze the usage of our system, we evaluate our system with: (1) 50 pieces of benchmark descriptions collected from users for popular shell commands; and (2) two use-case scenarios that involve participants with different scripting skill levels – and then we provide our observations drawn from them for further enhancements of the implementation.

RELATED WORK

For decades, researchers have constantly developed new tools and theories for computer science education, due to the rapid pace of computer adoption, the constant product and update cycles, and the significance of computing in the modern day world. Mainstream researchers suggest that the difficulties in teaching concepts in programming can be overcome with better tools. Some tools graphically visualize code (Guo, 2013; Orsega, Zanden, & Skinner, 2012) to simplify the way students understand programming and inspire their interests. Other tools focus on good scenario design, or set up efficient tasks that encompass all the major concepts (Wolz et al., 2009; Gallant & Mahmoud, 2008). For instance, game-based scenarios are excellent for engaging students of programming, as well as making the learning more fun (Lee, 2013; Basawapatna, Koh, & Repenning, 2010). In this paper, we take the next step to build a programming teaching system based on conversational interactions.

Researchers also argue about which language is best for beginners. Sattar and Lorenzen believe that the language Alice is best, since one codes by selecting choices from a drop down list which results in no syntax errors (Sattar & Lorenzen, 2009). Bagert, however, favors Ada for it's robustness

and it's elegant design (Bagert, 1998). In recent times, many researchers claim that Python is the best choice because of its simplicity and ubiquity (Alshaigy, 2013; Guo, 2013). We aim to build more languages into our system after our first trial, which was based on the LOGO programming language (Liu & Wu, 2014). We built the programming in ELIZA (PiE) to automatically synthesize LOGO programs, to introduce the process of creating algorithms to the uninitiated. In the research reported here, we chose the Shell, because of the universal importance of shell scripting and its automation capabilities, and to broaden the range of people that shell scripting is accessible to.

Another reason we choose shell scripting is the low cost of cognitive load during the learning process. Cognitive load theory (Sweller, 1988) uses the knowledge on human cognitive architecture to design the instructional procedures. One of the critical aspects of human cognition that is critical to the procedure design is that cognitive capacity in working memory is limited (De Jong, 2010). And these factors are main constraints for people's learning behavior and ability during a learning process. However, studies conducted by Sweller and his colleagues (Mwangi and Sweller, 1998; Sweller and Cooper, 1985) have shown that example-based learning (with interspersed problems to be solved) is more effective than learning only by problem solving. Natural Shell will provide a set of examples for popular system operations, e.g. massive copy, file searching, that users are familiar with. Thus, they can get a smooth transition from daily computer operations to system programming.

Interactivity is the last thing we focus on. From the practical educational perspective, in class, interactions are classified in three levels by Moore (1989): the learner-content, learner-instructor, and the learner-learner. Our tool encourages more interactions between the learners and their learning objectives, at the same time, it creates more opportunities for students to try commands and learn from their mistakes. To simulate that scenario in the process of coding, we adopt natural language dialog as the interface. In this case, students can immediately start giving instructions to the computer just as traditional seasoned programmers do, but in a rapid uptake that is unmatched by any previous method. On the other hand, MOOC (massive open online course) has been popular, but where the interaction styles between instructors and students are changing. Compared with traditional in-class education, co-located interactions are replaced by asynchronized communication (Nath and Agarwal, 2014), in which case, making the leaning process less efficient and fun. Natural Shell, which is an agent-based interactive programming system, can play the role as a responsive learning content and the instructor as well. By providing students the closest commands or making recommendations for incomplete instructions, students will have a better experience when interacting with our tool and their understanding on system functions can be enhanced during the learning process.

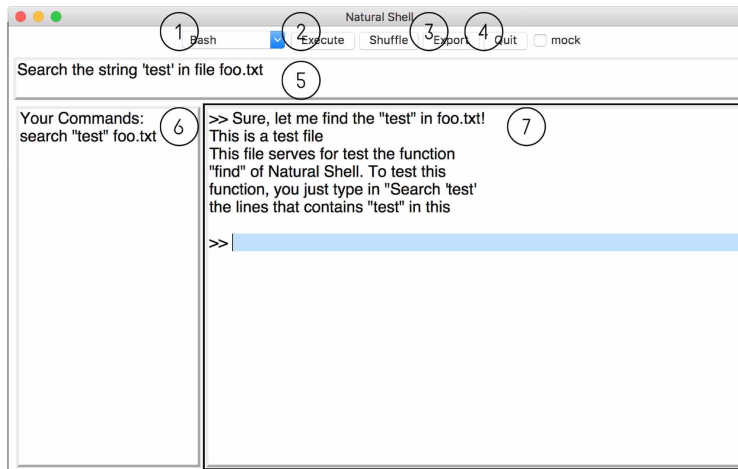
USER INTERFACE

Natural Shell is a new interface for users to interact with system kernel functions, within the design of a local desktop application. Instead of the numerous command line tools that have a dark background and over-simplified imperative functionality, Natural Shell is more user-friendly, with an interface more akin to a modern application program, as can be seen in Figure 2. There are three main boxes in our design, which are the Natural Command Box, Uni-Shell Command Box and Execution Result Box. In this section, we will explain the functionalities of this tool in detail.

Figure 2 shows a screen shot of Natural Shell. On the top of the entire interface are a few buttons that trigger functions. Before a user starts to type in the commands, he/she will first choose the desired target syntax among three mainstream syntaxes we provide: Bash, Csh and MS-DOS. The chosen option comes into immediate effect; however, the user can change it anytime during the interaction.

There are two modes of operation: Novice mode and Apprentice Mode. For novices who are not familiar with any shell syntaxes, Natural Shell enables them to harness the system functions using natural language commands. In this mode, the "natural command box" accepts the user's natural language descriptions, which either can be a single-line command or multi-line scripts. Unlike the original shell commands, there is no syntax restriction on the natural language inputs, which provides

Figure 2. User interface of natural shell. (1) Syntax Button: Choose the target syntax. (2) Execute Button: Try out your natural language command. (3) Export Button: Export the commands in target syntax as a local script. (4) Quit Button: Quit the application. (5) Natural Command Box: Type in your natural language commands here. (6) Uni-Shell Command Box: The synthesized commands are shown here. (7) Execution Result Box: Both the natural language feedback and execution return values are shown here.



users more flexibility in composing commands and generates fewer errors as well. After typing in their natural language commands, the user can expect: (1) commands in the Natural Shell syntax which are presented in the "Uni-Shell command box"; (2) natural language feedback that confirms the entered commands which is shown in the "execution result box"; and (3) return values from execution of the synthesized script commands which is also shown in the "execution result box". However, some users may have gained familiarity with Uni-Shell and it may be redundant for them to start with natural language descriptions. Therefore, in the Apprentice Mode, these users can directly compose their script in Uni-Shell syntax inside the "Uni-Shell command box" and they can either execute the script in Natural Shell or export the script to local storage in any target syntax.

KEY TECHNIQUES

In this section, we introduce the process of command synthesis with a motivating example. We also detail four key techniques adopted by Natural Shell within the process.

Example

Creating and deleting files are routine operations for end-users. Writing scripts to automate these tedious tasks saves time and labor. Unfortunately, with regard to online forums for scripting language beginners (Unity, 2011; Roblox, 2013; Siri, 2010), many users consider scripting difficult and would rather manually repeat the tasks than to compose a repeatable script with several lines. Many of them mention that mastering the rigid syntax of scripting languages is difficult. However, with the tool we proposed, learning to script can be relatively simplified. Consider the following operations that a user may conduct: do the following for every sub-folder: go into the sub-folder, select files that contain "foo" in the name and create a new directory called "bar", move the selected files into the new directory, go back to the parent folder. Based on the natural language descriptions, we wish to generate shell commands in both UNIX shell (Bash, Csh) and MS/DOS syntaxes, respectively, denoted as S_1, S_2.

Let's suppose the natural language description goes like this: "Repeat these for every sub-folder:" "create a new folder called 'bar' in the current directory." and "move the files containing 'foo' in the name into the new directory". Natural Shell processes the commands one by one. Then these

descriptions are decomposed into three separated commands. When a command in natural language is received by the ELIZA module, we tokenize the sentence phrase by phrase, denoted w_1, w_2, ..., w_n. After the phrase separation, we analyze the syntax of each sentence with the assistance from the ELIZA module. In this module we tag these phrase chunks with some predefined Tokens, such as the "Predicate", the "Variable", the "Control", or the "Redundant" by adopting regular expression matching. For instance, if all the descriptive sentences in the example are analyzed, the compositions are tagged as follows:

"Repeat these for every sub folder":

(Predicate)+(Redundant)+(Control)+(Variable)

"Create a new folder called 'bar' in the current directory":

(Predicate)+(Constant)+(Predicate)+(Variable)+(Variable)

"Move the files contains 'foo' in the name into the new directory":

(Predicate)+(Variable)+(Predicate)+(Regular)+(Redundant)+(Variable)

To understand sentences constructed by tokens, we design a couple of rules which are represented in the form of regular expression combinations. We will explain the rationale of choosing regular expression to express rules in the Keywords vs. Regular Expression Section. After one sentence is accepted by our system, the corresponding commands in Uni-Shell syntax are synthesized. We detail the design of this intermediate language in the Uni-Shell Section. To improve the accuracy of the transcription, rules are designed separately for each class of command. A class is defined as those descriptions that share the same predicates or control flows. Thus, descriptions in one class certainly have the same operator or control flow.

Table 2 shows the relationship between some function words and the Uni-Shell commands. In order to make the system more robust, that is, to enable the system to accept natural language as flexibly as possible, we collect a group of natural language commands for each Operator. For example, to represent the idea of "Remove", users can choose words like "remove", "delete", "take away" or "move away". These words or phrases are considered to be synonymous in natural language, thus carrying the same semantic meaning. Different verbs in natural language are used in these sentences but the sentence structure is the same or very similar. Under this circumstance, we adopt transform

Table 2. Functions words and Uni-Shell commands

Function Words	Uni-Shell Commands
Change directory	Direct to [path]
Rename	Rename [File/Directory] to [name]
Remove	Remove [File/Directory]
Create	Create [Directory]
Change mode	+/-[File/Directory] [read/write/execute]
Repeat	For [Var] in [Range] Do [CMD]
While	While [COND] Do [CMD]

T which is borrowed from the ELIZA system to categorize all predicates or parameters with the same meaning, into a dedicated term. We base our system on the 1987 Penguin edition of Roget's Thesaurus of English Words and Phrases (Roget, 1982) for the synonyms to maintain a dictionary of words that can be transformed to common terms.

In the last step, the commands in Uni-Shell syntax are ported to scripts in any of the assigned target languages. Adapted to the shell or batch files used in the current system, shell scripts or Windows batch files are generated from the Uni-Shell commands. Table 3 shows the Uni-Shell commands and the corresponding Bash, Csh, and MS/DOS commands. To show the versatility of Uni-Shell, we pick up the commands, the syntaxes of which are different among the three. Before execution, to ensure the accuracy of these shell commands, we also designed a static type checker to verify the validity of each parameter that was synthesized from the natural language descriptions. The type checker automatically casts the parameters according to pre-defined types, e.g., the parameter represents an existing file pattern should match an existing file name in the working path. Detailed specifications about this checker are introduced in the Static Type Checker Section.

Uni-Shell

"Uni-Shell" is a context-free language defined by a new shell syntax which has more natural features than system-specific shells, and such features make each command more readable and easier to learn. Take the command "copy" in MS/DOS or the "cp" in UNIX shell as an example. To illustrate this command in "Uni-Shell ", we adopt the syntax "copy [file/directory] from [directory] to [directory] " in Uni-Shell. Besides the increased ease in reading it, "Uni-Shell" is an abstracted universal shell that can be ported to any other shell programming languages, beyond the three presented in this

Table 3. Uni-Shell and the corresponding bash, csh, and batch commands

Uni-Shell	Bash	Csh	Batch
Rename [File/Directory] to [name]	mv [s] [d]		RENAME [s] [d]
Remove [File/Directory]	rm [option] [File/Dir]		DEL/DELTREE [File/Dir]
Match [r.e.] [FILE]	grep [pattern] [FILE]		FIND [pattern] [FILE]
+/-[File/Directory] [r/w/e]	chmod [option][+/-][x/r/w] [File]		ATTRIB [+/-][r/a/s/h] [File]
	if COND1; then	if COND1; then	if COND1 CMD1 else CMD2
	CMD1	CMD1	
If [COND1] Then [CMD1] Else [CMD2]	else	else	
	CMD2	CMD2	
	fi	end if	
	for [VAR] if LIST; do	for [VAR] if LIST; do for	for [variable] in [LIST] do CMD
For [Var] in [Range] Do [CMD]	CMD	CMD	
	done	end	
	while COND; do	while(COND)	:loop
While [COND] Do [CMD]	CMD	CMD	CMD
	done	end	if COND goto loop

paper. Despite the increased learnability of "Uni-Shell", it is not as redundant as the natural language descriptions for the same tasks. In addition to conducting the natural language conversations with the system, users can also write platform-independent scripts using the new "Uni-Shell".

Keywords vs. Regular Expression

We have designed a set of rules constructed by regular expressions to map descriptions in natural language to Uni-Shell Commands. We collect a group of natural language synonyms for each Operator. To match the descriptions in natural language, we adopted regular expressions instead of keywords. ELIZA is a prototype chatbot that plays the role as a therapist (Weizenbaum, 1966). To understand the sentences in natural language, that is English, after the word decomposition, ELIZA uses a keyword mechanism. This keyword mechanism is very intuitive and can filter out target sentences and then respond with the corresponding answers. However, the rules based on keywords are not as efficient as the rules constructed with regular expressions. Users misspell terms frequently, for example the misuse of tense is quite common. To match the multiple descriptions from users, the number of rules increases linearly since there is at least a verb in each sentence. With the increasing number of rules, the difficulty in both testing and adding more rules arise. But number can be dramatically reduced when we adopt regular expressions to construct the rules.

Static Type Checker

In our pilot study, we noticed that some descriptions are successfully compiled by Natural Shell. However, the synthesized target commands are still problematic. Observing the data, we found that two major reasons causing errors are parameter formats and rule conflicts. Because there are no limitations upon input descriptions, the parameter formats are scattered from case to case. For example, when the user tries to mention a file name, some descriptions with special identifiers like "(foo.txt)" and "<foo.txt>" will be adopted. For most cases, these special identifiers are successfully caught by the regular expressions, yet some other cases still exist. The other problem is that some rules conflict. Since the order of rules is fixed, a sentence will be decoupled with the first rule that can be applied to it. This mechanism caused some error cases, such as the rules for "whereis" and "find". The rules for these two operators are quite similar and the only difference is the object to be found. While "whereis" is used for searching files related to an executable utility, "find" is used for searching files in general, but within a targeted directory structure.

To improve the performance of Natural Shell, we implemented a basic static type checker for the synthesized commands. The type checker is deployed with the syntax directed translation scheme, since we can get the parsed syntax tree of synthesized command from the SiE module. In this type system, we have defined 8 types and to make it scalable, the implementation is separated from the SiE script. Similar to classic type checkers, we have implemented our checking rules as shown in Table 4 and casting rules as shown in Table 5 which helps us to resolve the errors resulted by scattered user format.

Type checker will first generate the typed Abstract Syntax Tree (AST) from synthesized commands and it then helps us to infer the type of a given parameter. The proposition $\Gamma \vdash v : \tau$ is a typing judgment stating that the term (program component) v has type τ under the type checking environment Γ, which is a mapping from variables to their types. Each type checking rule has a set of premises above the bar and a conclusion under the bar. The type checker will check the correctness of a synthesized shell program starting with the rule (T-PROG) with an empty environment. The type checker will progress unless all competing premises hold, in which case some other rules will be employed to verify the correctness of each premise. The system will directly cast the parameter to its correct type according to the type casting rules in Table 5. For example, if the command "match data <foo.txt>" is generated from SiE script, we first check the rule (T-PROG) and it should hold if "match data <foo.txt>" is typed Cmd. With the rule (T-MATCH), we can verify the type of this commands when the two parameters are of the correct type, such that (data: String) (<foo.txt>: File). With the casting methods as shown in Table 5 for String and File, the command will be formatted as "match

Table 4. Type checking rules for some shell commands

$$\frac{\Gamma \vdash e_1 : \mathit{File} \quad \Gamma \vdash e_2 : \mathit{File} \quad e_1 \neq e_2}{\Gamma \vdash \mathit{rename}\ e_1\ e_2 : \mathit{Cmd}}\ \text{(T-RENAME)} \qquad \frac{\Gamma \vdash e_1 : \mathit{String} \quad \Gamma \vdash e_2 : \mathit{File}}{\Gamma \vdash \mathit{match}\ e_1\ e_2 : \mathit{Cmd}}\ \text{(T-MATCH)}$$

$$\frac{\Gamma \vdash e : \mathit{Directory}}{\Gamma \vdash \mathit{direct\ to}\ e : \mathit{Cmd}}\ \text{(T-CD)} \qquad \frac{\Gamma \vdash e : \mathit{Directory}}{\mathit{create}\ e : \mathit{Cmd}}\ \text{(T-CREATE)} \qquad \frac{\Gamma \vdash e : \mathit{File}}{\mathit{search}\ e : \mathit{Cmd}}\ \text{(T-SEARCH)} \qquad \frac{v \in \mathit{dom}(\Gamma)}{\Gamma \vdash v : \Gamma(v)}\ \text{(T-VAR)}$$

$$\frac{\Gamma \vdash \mathit{Hd} : \tau \quad \Gamma, v : \tau \vdash \mathit{Cmd} \quad \Gamma \vdash \mathit{for}\ [v]\ \mathit{in}\ \mathit{Tl}\ \mathit{do}\ [c] : \mathit{Cmd}}{\Gamma \vdash \mathit{for}\ [v]\ \mathit{in}\ [\mathit{Hd} : \mathit{Tl}]\ \mathit{do}\ c : \mathit{Cmd}}\ \text{(T-FOR 1)} \qquad \frac{\Gamma \vdash c_1 : \mathit{Cmd} \quad \Gamma \vdash c_2 : \mathit{Cmd}}{\Gamma \vdash c_1 ; c_2 : \mathit{Cmd}}\ \text{(T-SEQ)}$$

$$\frac{}{\Gamma \vdash \mathit{for}\ [v]\ \mathit{in}\ []\ \mathit{do}\ c : \mathit{Cmd}}\ \text{(T-FOR 2)} \qquad \frac{\Gamma \vdash e : \mathit{Boolean} \quad \Gamma \vdash c_1 : \mathit{Cmd} \quad \Gamma \vdash c_2 : \mathit{Cmd}}{\Gamma \vdash \mathit{if}\ e\ \mathit{then}\ c_1\ \mathit{else}\ c_2 : \mathit{Cmd}}\ \text{(T-IF)} \qquad \frac{\Gamma \vdash e : \mathit{File}}{\mathit{remove}\ e : \mathit{Cmd}}\ \text{(T-REMOVE)}$$

$$\frac{\Gamma \vdash e_1 : \mathit{File} \quad \Gamma \vdash e_2 : \mathit{Mode}}{\Gamma + \ e_1\ e_2 : \mathit{Cmd}}\ \text{(T-CHANGE MODE 1)} \qquad \frac{\Gamma \vdash e : \mathit{Executable}}{\mathit{whereis}\ e : \mathit{Cmd}}\ \text{(T-WHEREIS)} \qquad \frac{\Gamma \vdash e : \mathit{GZFile}}{\mathit{ungz}\ e : \mathit{Cmd}}\ \text{(T-UNGZ)}$$

$$\frac{\Gamma \vdash e_1 : \mathit{Directory} \quad \Gamma \vdash e_2 : \mathit{Mode}}{\Gamma + \ e_1\ e_2 : \mathit{Cmd}}\ \text{(T-CHANGE MODE 2)} \qquad \frac{\Gamma \vdash e : \mathit{TARFile}}{\mathit{untar}\ e : \mathit{Cmd}}\ \text{(T-UNTAR)} \qquad \frac{\vdash c : \mathit{Cmd}}{\emptyset \vdash c : \mathit{Prog}}\ \text{(T-PROG)}$$

Table 5. Type casting rules

Type	Casting Rules
String	Cast from any format to a string with double quotes.
Integer	Cast from any format to an integer number.
Executable	Check whether it is a piece of executable.
Mode	Check whether it is in {read, write, execute}.
File	Auto-complete the name referred to by existing files.
Directory	Auto-complete the name referred to by existing folders.
GZfile	Check if end with .gz and auto-complete the name referred to by existing files.
TARfile	Check if end with .tar and auto-complete the name referred to by existing files.

'data' foo.txt". On the other hand, with this mechanism, Natural Shell can solve conflict rules based on given parameters and their corresponding types. For example, from the user's description: "Find where is foo.txt", two commands can be generated by Natural Shell: "find foo.txt" and "whereis foo.txt". However, since "foo.txt" is not valid executable, the synthesized result "whereis foo.txt" will be ruled out by the type checker.

Recommendation vs. Manual

When a programmer fails to compose a shell command in the correct syntax, the system terminal will always return the usage which is normally in an abbreviated format. For example, if one mistakenly typed in a "ping" command in terminal, what is expected is the usage information from manual as follows:

```
ping [−A aCDdfnoQqRrv][−b boundif][−c count][−G sweepmaxsize][−g sweepminsize]
[−h sweepincrsize][−i wait][−l preload][−M maskltime][−m ttl][−P policy]
[−p pattern][−S src_addr][−s packetsize][−t timeout][−W waittime][−z tos] host
ping[−A aDdfLnoQqRrv][−b boundif][−c count][−I iface][−i wait][−l preload]
[−M maskltime][−m ttl][−P policy]
[−p pattern][−S src_addr][−s packetsize][−T ttl][−t timeout][−W waittime][−z tos] mcast−group
```

In this abbreviated format, all possible options for this command are provided for reference. However, there is no ranking in the order of these options, in which case, before the user tries to get a correct syntax, they have to figure out which option they want. In our design, to reduce the user's effort in such command option selection, Natural Shell recommends template commands based on your semantics, if it does not find a rule that can process your natural language description. For example, if a user wants to use the ping command and says "ping 5 times" to Natural Shell, since there is no such rule in our system that can handle this description, it will match with the most similar rules and return the corresponding template commands to the user. As shown in Figure 3, two command templates are provided: "ping [host] [how long] timeout" and "ping [host] [how many] packets". Then, if the user realizes he wants to play the latter one and he types a command in the correct syntax to Natural Shell, he will then get the expected result from the Natural Shell terminal.

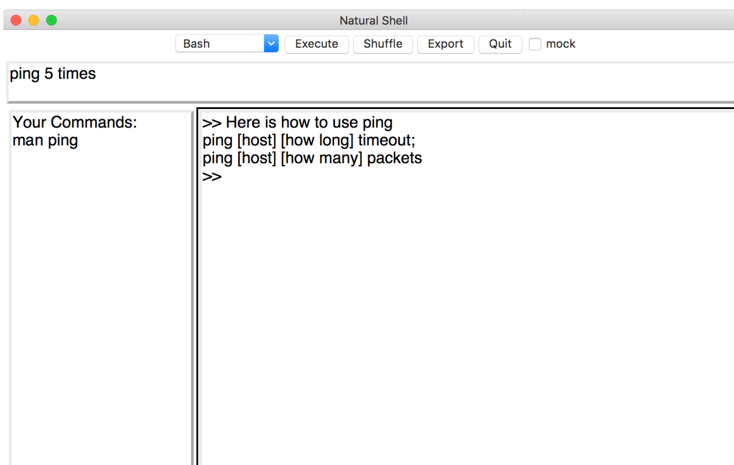
Embedded Terminal

Since our system provides an interface for users to interact with system kernel functions using natural language descriptions, there is also an embedded terminal for the results returned. This is similar to traditional command line terminals in Unix/Linux and Windows systems. The embedded terminal inherits the interactivity of the underlying original terminal, which returns the results immediately after the execution. However, in addition to the returns from system calls, the embedded terminal returns natural language feedback to users, as a confirmation of their original command description. For novice users, this confirmation plays a critical role in the learning process in two ways: it restates the users descriptions in a comprehensible but more formal way to validate the semantics of the desired commands; and secondly, it will assist with error instructions when the descriptions are ambiguous or unclear.

OVERCOME END-USER FRUSTRATIONS

Some difficulties in composing scripts encountered by users are quite common. By observing the questions from a UNIX shell scripting online forum (UNIX, 2015) we conclude that most user-posted questions on syntax can be categorized into two types: overwhelming options, and nuances between Shells. In this section, we will elaborate on how our solution overcomes these user frustrations.

Figure 3. Recommend templates for use of the "ping" command



Option

In the existing scripting languages including Bash, Csh and MS/DOS, options are essential parts of each command to differentiate the various functionalities. But to users who are not familiar with the command syntax, they quickly become overwhelming. Although the options are usually abbreviations for some English words, in many cases, people often forget them even after recurrent practice, since the number of options is large. In addition, an option with the same name can have the different meanings across different commands. For example, in bash, "-f" means suppress most error messages in "rm" and "chmod". However, "-f" indicates that the pattern is from a file when used with "grep". When a user starts to learn a shell from scratch, she often finds these inconsistent uses of option names, confusing and misleading.

However, with Natural Shell, such problems with "options" are avoided. Assisted by our tool, people may use a high-level abstracted sentence to express a command. The system then automatically generates the corresponding option, based on the semantic meaning of the sentence. For example, by saying "delete the files in this folder without confirmation", our tool will recognize the keyword "without confirmation" and translate it into "-f". Compared to the short and cryptic command representations adopted in the classic shell designs, natural language input makes our interface friendlier, and enables users to write scripts intuitively, without the need to check the manual for the options.

Nuance Between Shells

We define Uni-Shell as an intermediate language that can be ported to the other shell commands, overcoming the incompatible syntaxes across competing shell systems. When a user is trying to write a script, she might check some commands via online manuals. It is interesting that the syntaxes on different platforms are similar but not exactly the same. She will be frustrated to find a newly-learned command is illegal when she switches to another platform. The human mind processes similar words or homophones into the one category and that contributes to why people often misspell a word, by choosing the wrong one (e.g. 'their' instead of 'there'). The same principle can also be applied to command syntaxes. Nuances between commands can lead to similar such confusion.

Our Uni-Shell performs well with wide-compatibility and better readability than native shells on multiple operating systems. The feature of the natural language descriptions that it understands can be both UNIX-like and Windows-like. For example, the slash is used differently between shell programming languages like Bash or Csh and the MS/DOS prompt. Furthermore, the file system in Windows is case-insensitive, but the opposite is true for UNIX/Linux. However, our Uni-Shell can tolerate both the Forward Slash and Back Slash and the insertion of blanks is handled as well. Also, it can recognize filenames in different case mode according to the current system. Another advantage of Uni-Shell is that, it makes the intention of the command more clear to the user. Take the "Copy" command as an example, Uni-Shell defines this command as "Copy from [source] to [destination]". This enhanced command is more meaningful to newbies since they often forget the order of [source] and [destination] in the native shell or the Windows prompt.

EVALUATION

We conducted two formative studies: a system evaluation to test the effectiveness of Natural Shell. We are interested in how the proposed characteristics of Natural Shell: 1) natural syntax, 2) interactivity, 3) cross-platform, will influence students' interest in learning system scripting and what is students' selection for learning and using shell scripting compared with original shell syntaxes.

System Effectiveness

In this section, we evaluate the effectiveness of Natural Shell in synthesizing the right commands in target shell syntaxes. We implemented Natural Shell in Python with a simplified interface built upon Tkinter (the defacto standard GUI for the Python language). To conduct the evaluation, we compiled a benchmark from popular commands gathered from online forums (Ramesh, 2010), and asked users to describe these commands in their own language. . To justify the benchmark tasks, we also recruited an outside expert who has 5+ years of scripting experience to select and generate our own benchmark with 50 tasks from these frequent commands. We recruited 4 users with different shell programming background ranging from novices who have no shell programming experience, to experts who can quickly compose shell scripts in both Bash and Csh. Overall, we collected 200 natural language descriptions for the 50 tasks (4 phrases for each) and we fed Natural Shell these descriptions with and without the type checker.

Of the 200 natural language phrases, 142 were correctly synthesized into underlying shell syntax by Natural Shell. That is, Natural Shell was able to synthesize 71% of real-life shell commands that it was designed for. In addition, we noticed that, among the 58 phrases that failed, 21 were parsed with the rules in Natural Shell but could not be executed successfully due to format errors or rule conflicts. After that, we turned on the incorporated type checker and run the test again. All 21 phrases were then correctly interpreted and executed and we concluded that the adoption of the type checker increases the overall accuracy of Natural Shell to 82%. The benchmark covers most of the popular commands used, which can be seen in Table 6. We denote the number passed with the type checker on, as Pass(T) and the ones without as Pass(NT).

Lab Study

In this section, we will describe the performance of our tool with a usability lab study. We recruited 10 college-aged teens (5 males and 5 females) for the one-on-one think-aloud lab study. Participants were brought into a lab individually with a researcher, and given a set of use-case scenarios that lead to tasks and usage of Natural Shell. Before the study, we let each of the participants briefly introduce themselves with some demographic information and we also let them evaluating themselves on programming and shell scripting experience from 1 (non-experienced) to 7 (familiar). During the study, we demonstrate the functionality of Natural Shell to synthesize the commands in the three target syntaxes as designed and then let them describe how they understand the tool. They were encouraged to play with Natural Shell from basic one sentence command to scripts within 10 lines as shown in Table 7. The average session time for the 10 participants is 23 minutes. We recorded their descriptions on their understanding of Natural Shell in the think-aloud session and the times they referred to the manual book and raised questions during the process as well.

Following the think-aloud lab study, we conducted an interview with each participant in depth about the research question we are interested in that how do students select their languages when given the opportunity to select the language they use to solve their scripting tasks. Students were given the opportunity to select the language they used to solve their problems, which could be completed with or without Natural Shell.

Struggles with Scripting

During the think-aloud session, participants struggles with composing a script on their own. All the 10 participants successfully executed the scripts as they want. However, most of the commands (231 of 301) they tried during their 23 minutes are basic shell commands such as copy, paste, and file move and only a few control flows are successfully constructed. Novice students (7 of 10), the intersection of who have no background in shell scripting and who rated themselves below 3 generated more questions and they referred more times to the manual book than the experienced students (3 of 10).

Table 6. Characteristics of the benchmark set extracted from online forums

	Benchmark Command Descriptions	Sum	Pass(NT)	Pass(T)
1	Extract from an existing tar archive.	4	4	4
2	Search for a given string in a file.	4	4	4
3	Search for a given string in all files recursively.	4	1	3
4	Find files using file-name.	4	2	4
5	Converts the DOS file format to Unix file format.	4	2	2
6	Add line number for all non-empty-lines in a file.	4	4	4
7	Remove duplicate lines.	4	2	4
8	Print only specific field from a file.	4	1	3
9	Go to the 143rd line of file.	4	4	4
10	Open the file in read only mode.	4	2	4
11	Find the differences between files.	4	2	4
12	Sort the files in ascending order.	4	4	4
13	Copy all images to external hard-drive.	4	3	3
14	Download all the URLs mentioned in the url-list.txt file.	4	3	3
15	Display file size in human readable format.	4	3	3
16	Print working directory.	4	3	3
17	To create a *.gz compressed file.	4	2	4
18	To uncompress a *.gz file.	4	0	4
19	Shutdown the system and turn the power off immediately.	4	4	4
20	Shutdown the system after 10 minutes.	4	4	4
21	Reboot the system using shutdown command.	4	4	4
22	Force the file system check during reboot.	4	3	3
23	View crontab entry for a specific user.	4	2	4
24	Synchronize the hardware clock with the system date.	4	4	4
25	Restart a service.	4	4	4
26	To view current running processes.	4	3	3
27	To displays only the processes that belong to a particular user.	4	3	3
28	Displays the file system disk space usage.	4	2	2
29	Kill command to terminate a process.	4	3	3
30	Get confirmation before removing the file.	4	2	3
31	Recursively removes all files directories under a directory.	4	4	4
32	Copy file1 to file2 preserving mode, ownership and timestamp.	4	3	3
33	Copy file1 to file2. Confirmation before overwrites.	4	2	2
34	Rename file1 to file2.	4	3	3
35	Prints the content of file1 followed by file2 to stdout.	4	3	3
36	Create a directory and mount it.	4	1	1

continued on following page

Table 6. Continued

	Benchmark Command Descriptions	Sum	Pass(NT)	Pass(T)
37	Give full access to user and group on a specific file.	4	2	2
38	Revoke all access for the group on a specific file.	4	2	2
39	Apply file permissions recursively to all files in sub-directories.	4	2	2
40	Change your password.	4	4	4
41	Remove password for a specific user.	4	4	4
42	Create a directory called temp.	4	2	2
43	View all the interfaces along with status.	4	2	2
44	Find out where a specific Unix command exists.	4	4	4
45	Search for the location of a specific file.	4	3	3
46	Print the last 10 lines of a file by default.	4	1	1
47	View huge log files.	4	4	4
48	Ping a remote host by sending only 5 packets.	4	4	4
49	View the system date.	4	4	4
50	Set the system date.	4	4	4
Total		200	142	163

Table 7. Interaction of the use-case scenario

User Transcript	Response from Natural Shell	Batch File
If there is no specified file name, end the script	> Okay, Let's end the scripts when no argument specified for the file names.	IF "%1" == "" GOTO END
Create a directory called SAVE	> Okay, Let's create a directory called SAVE.	MD SAVE
Label the following as MOVE-FILE	> Okay, Let's label this part as MOVE-FILES	:MOVE-FILES
Copy the specified files to the SAVE folder	> Okay, Let's copy the files specified to the folder called SAVE.	XCOPY %1 SAVE
Shift to the next specified file	> Okay, Let's consider the next file.	SHIFT
If no files specified, then go to FILE-DELETE	> Okay, Let's do FILE-DELETE of no files specified	IF "%1"=="" GOTO FILE-DELETE
Else, go back to MOVE-FILE	> Okay, else we go back to the MOVE-FILES.	GOTO MOVE-FILES
End this part	> Okay, Let's end MOVE-FILES.	
Label the following as FILE-DELETE	> Okay, Let's label this part as FILE-DELETE.	:FILE-DELETE
Delete all the files in current folder	> Okay, Let's delete all files in current folder.	DEL
End the FILE-DELETE	> Okay, Let's end FILE-DELETE.	
Please move all the files in SAVE to current folder	> Okay, Let's move all files in folder SAVE to current folder.	MOVE SAVE
Delete the directory called SAVE	> Okay, Let's delete the folder called SAVE.	RD SAVE
Label here as the END	> Okay, Let's label here for the ending.	:END

From the think-aloud transcripts, we got some interesting findings. Since Natural Shell adopts natural language commands, every student started to try the tool within 1 minute after the introduction. They described the tool easy to start with and to understand what is shell scripting was easier in this way. They find the recommendation function of Natural Shell very useful and tried a lot incomplete commands as they wanted. The recommended feedbacks were easy to understand as well. However, according to most novice students (4 of 7) who had no programming background before the study, they can compose a script with a for-loop and they can delete a folder with *–r* but still did not understand the relationship between iteration and recursion. As an explanation, this is a trade-off between natural syntax and low-level semantics. To achieve the nature in syntax, many of the implementation details are hidden.

Selection for Natural Shell

In the interview session, most students (9 of 10) expressed their willing to be assisted with Natural Shell than scripting using the original system syntax. They voted for Natural Shell for three reasons, simplicity, easy to remember and cross-platform. The only one student who did not vote for Natural Shell is an experienced programmer and he is very familiar with Bash scripting (5 years' experience). As he commented, the tool will make it more complicated for him to compose a script compared with his original habits.

(Simplicity) One of the students commented on Natural Shell as "Toy Scripting". She just started to learn Java has no experience with shell scripting. She said, "I will never use shell scripting without Natural Shell because the original syntax is too complex". From her talk, we can conclude that the simplicity of Natural Shell commands will be the most reason that she will try scripting or she would prefer conducting the system tasks without a script. Her voice represents most of the non-experienced participants in our study and we believe Natural Shell can encourage students to try system scripting which will enhance their daily working efficiency.

(Easy to remember) All the 9 students who voted for Natural Shell expressed their feeling about the syntax of Natural Shell. On student commented that "It is great to know that there is no syntax for a programming language. According to my previous experience, only how to print 'hello world' can be remembered during the first round going through the manual book of any programming languages." As described by these students, Natural Shell will assist their experience with scripting in the first a few weeks as an introductory. It will be easier for novices to start with compared with any syntax provided by the system.

(Cross-platform) Only 2 students mentioned the functionality of Natural Shell as we introduced which is cross-platform while the other students just asked questions like, "I am using OSX. Can I still use this tool on my platform"? One of the two students who mentioned this advantage of Natural Shell also expressed his opinions that this kind of natural syntax will be more useful if ported to any other programming languages.

CONCLUSION

Scripting is widely-used for automating the execution of tasks. Despite its popularity, it remains difficult to use for both beginners and experts. Beginners are mostly overwhelmed by the numerous cryptic commands, while experts struggle with incompatible syntaxes across different systems, including complicated reuse of the option names for different purposes in those different Shells. Our solution to this problem is a system we built called Natural Shell. It transforms natural language descriptions into shell scripts. In the process, as a first step within Natural Shell, it translates the natural language input from a user, to an intermediate level script we call Uni-Shell, which is independent of the various underlying native scripts. In the last step, Natural Shell, transforms the Uni-Shell script into the appropriate underlying native script, executes it, then returns the results to the user.

We implemented a sampling of the most commonly used commands from real world shells into our system, and conducted qualitative user studies on several users with different scripting skill levels. We tested our system with 200 samples of natural language phrases that call upon 50 popular underlying shell commands, and Natural Shell successfully synthesized 82% of them. The preliminary results are promising and encouraging. We also conducted a small-scaled lab study with a think-aloud session and an interview session. Participants' feedbacks were positive on Natural Shell and most of them considered our tool a better selection comparing with original shell syntaxes. As a first step in evaluating Natural, the shell command sampling pool was significant, but the user study was quite limited. The user-study was qualitative in nature, with a view to getting preliminary end-user feedback on the prototype system. In the future, we plan to continue to develop the system to synthesize more commands, and run a larger scale quantitative user study to collect more user feedback, and a statistically significant quantity of end-user data.

REFERENCES

Alshaigy, B. (2013). Development of an interactive learning tool to teach python programming language. *Proceedings of the 18th ACM conference on innovation and technology in computer science education* (pp. 344–344). doi:10.1145/2462476.2465601

Bagert, D. J. (1998). Using Ada to teach programming language design concepts. *ACM SIGAda Ada Letters*, *18*(1), 54–64. doi:10.1145/280495.280499

Ballard, B. W., & Biermann, A. W. (1979). Programming in natural language: NLC as a prototype. In *Proceedings of the 1979 annual conference* (pp. 228–237). New York, NY, USA: ACM. doi:10.1145/800177.810072

Basawapatna, A. R., Koh, K. H., & Repenning, A. (2010). Using scalable game design to teach computer science from middle school to graduate school. *Proceedings of the fifteenth annual conference on innovation and technology in computer science education* (pp. 224–228). doi:10.1145/1822090.1822154

Brill, E. (1992). A simple rule-based part of speech tagger. *Proceedings of the workshop on speech and natural language* (pp. 112–116). doi:10.3115/1075527.1075553

De Jong, T. (2010). Cognitive load theory, educational research, and instructional design: Some food for thought. *Instructional Science*, *38*(2), 105–134. doi:10.1007/s11251-009-9110-0

Dijkstra, E. W. (1979). On the foolishness of "natural language programming". In *Program construction* (pp. 51–53). Springer. doi:10.1007/BFb0014656

Gallant, R. J., & Mahmoud, Q. H. (2008). Using Greenfoot and a moon scenario to teach java programming in cs1. *Proceedings of the 46th annual southeast regional conference* (pp. 118–121). doi:10.1145/1593105.1593135

Guo, P. J. (2013). Online python tutor: Embeddable web-based program visualization for cs education. *Proceeding of the 44th ACM technical symposium on computer science education* (pp. 579–584). doi:10.1145/2445196.2445368

Hazzan, O., Lapidot, T., & Ragnis, N. (2011). *Guide to teaching computer science: An activity-based approach* (2nd ed.). London: Springer. doi:10.1007/978-0-85729-443-2

Ismal, M. N., Ngah, N. A., & Umar, I. N. (2010). Instructional strategy in the teaching of computer programming: a need assessment analyses. *The Turkish Online Journal of Educational Technology, 9*(2), 125-131.

Lee, M. J. (2013). How can a social debugging game effectively teach computer programming concepts? *Proceedings of the ninth annual international ACM conference on international computing education research* (pp. 181–182). doi:10.1145/2493394.2493424

Lee, W. I., & Lee, G. (1995). From natural language to shell script: A case-based reasoning system for automatic unix programming. *Expert Systems with Applications*, *9*(1), 71–79. doi:10.1016/0957-4174(94)00050-6

Liu, X., & Wu, D. (2014). PiE: Programming in ELIZA. *Proceedings of the 29th IEEE/ACM international conference on automated software engineering* (ASE).

Moore, M. G. (1989). Editorial: Three types of interaction. *American Journal of Distance Education*, *3*(2), 1–7. doi:10.1080/08923648909526659

Mwangi, W., & Sweller, J. (1998). Learning to solve compare word problems: The effect of example format and generating self-explanations. *Cognition and Instruction*, *16*(2), 173–199. doi:10.1207/s1532690xci1602_2

Nath, A., & Agarwal, S. (2014). Massive Open Online Courses (MOOCs)–A comprehensive study and its application to green computing in higher education institution. *International Journal (Toronto, Ont.)*, *2*(2), 7–14.

Orsega, M. C., Vander Zanden, B. T., & Skinner, C. H. (2012). Experiments with algorithm visualization tool development. *Proceedings of the 43rd ACM technical symposium on computer science education* (pp. 559–564).

Ramesh, N. (2010). *50 Most Frequently Used UNIX / Linux Commands (With Examples).* Retrieved from http://www.thegeekstuff.com/2010/11/50-linux-commands/?utm_source=feedburner

Ramirez987. (2015). *UNIX shell programming forum.* Retrieved from http://www.unix.com/shell-programming-and-scripting/

Rist, R. S. (1991). Knowledge creation and retrieval in program design. *Human-Computer Interaction, 6, 1e46.*

Robbins, A., & Beebe, N. H. (2005). *Classic shell scripting*. O'Reilly.

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: a review and discussion. *Computer Science Education, 13(2), 137e172*

Roblox. (2013). Why is scripting so complicated? Retrieved from http://roblox.wikia.com/wiki/Thread:52908

Roget, P. M. (1982). *Roget's thesaurus*. Longman Group Limited.

Sattar, A., & Lorenzen, T. (2009). *Teach Alice programming to non-majors*. ACM SIGCSE Bulletin.

Siri, L. (2010). Friends don't let friends program in shell script. Retrieved from https://www.turnkeylinux.org/blog/friends-dont-let-friends-program-shell-script

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. *Cognitive Science*, *12*(2), 257–285. doi:10.1207/s15516709cog1202_4

Sweller, J., & Cooper, G. A. (1985). The use of worked examples as a substitute for problem solving in learning algebra. *Cognition and Instruction*, *2*(1), 59–89. doi:10.1207/s1532690xci0201_3

Unity. (2011). Why is scripting so hard? Retrieved from http://answers.unity3d.com/questions/54664/why-is-scripting-so-hard.html

UNIX. (2015). Unique entries for multiple files. Retrieved from http://www.unix.com/ shell-programming-and-scripting/260516-unique-entries-multiple-files.html

Weaver, G. A., & Smith, S. W. (2012). Xutools: Unix commands for processing next-generation structured text. In LISA.

Weizenbaum, J. (1966). ELIZA—a computer program for the study of natural language communication between man and machine. *Communications of the ACM*, *9*(1), 36–45. doi:10.1145/365153.365168

Wolz, U., Leitner, H. H., Malan, D. J., & Maloney, J. (2009). Starting with scratch in CS 1. In *ACM*. *SIGCSE Bulletin*, *41*(1), 2–3. doi:10.1145/1539024.1508869