STM JOURNALS

Journal of
Advances in Shell Programming

ISSN: 2395-6690
Volume 11, Issue 3, 2024
September–December
DOI (Journal): 10.37591/JoASP

https://journals.stmjournals.com/joasp

Review JoASP

The Power of the Shell: A Deep Dive into Shell Programming Techniques

V. Basil Hans

Abstract

Shell programming is a fundamental skill in Unix/Linux environments, enabling users to automate tasks, streamline workflows, and enhance system efficiency. This paper, "The Power of the Shell: A Deep Dive into Shell Programming Techniques," explores the core concepts, commands, and strategies that make shell scripting a powerful tool for both novice and advanced programmers. By examining key elements such as control structures, file manipulation, process management, and error handling, the paper highlights how shell scripts can be used to solve complex problems, automate repetitive tasks, and manage system resources. Furthermore, it delves into advanced techniques, including the use of regular expressions, pipes, and filters, demonstrating how shell scripting can be harnessed to optimize performance in various computing environments. This comprehensive overview provides readers with the knowledge and practical insights necessary to effectively utilize shell programming for automation, system administration, and beyond.

Keywords: Shell scripting, automation, Unix/Linux, system administration, command-line techniques

INTRODUCTION

Shell programming, a cornerstone of the Unix and Linux environments, plays a pivotal role in the automation of tasks, simplification of system management, and enhancement of overall efficiency. As operating systems evolve, the demand for efficient and adaptable methods to interact with these systems has become increasingly important. The Unix shell serves as both an interactive command interpreter and scripting language, allowing users to interact directly with the operating system through commands and scripts [1].

At its core, shell scripting allows users to automate repetitive tasks, manage files, handle processes, and configure systems using simple text-based scripts. Unlike traditional programming languages, shell scripts are designed to be executed directly by the system's command-line interface, providing a lightweight and efficient method to handle both small-scale and complex operations.

This paper, *"The Power of the Shell: A Deep Dive into Shell Programming Techniques,"* seeks to uncover the full potential of shell scripting by examining the key programming constructs, commands, and techniques. Through this exploration, we aim to demonstrate how shell programming can be leveraged to improve system performance, streamline operations, and solve complex computing problems. From basic command usage to advanced automation strategies, this study provides a comprehensive view of how shell programming remains a powerful tool in modern computing.

***Author for Correspondence**
V. Basil Hans
E-mail: vhans2011@gmail.com

Research Professor, Department of Management and Commerce, Srinivas University, Mangaluru, Karnataka, India

Received Date: October 28, 2024
Accepted Date: October 29, 2024
Published Date: November 04, 2024

Citation: V. Basil Hans. The Power of the Shell: A Deep Dive into Shell Programming Techniques. Journal of Advances in Shell Programming. 2024; 11(3): 41–50p.

Learning shell programming is vital for any system administration, programmer, or power user. It comprises a command-line interpreter that

permits the interactive use of commands to perform specific tasks [2]. The command-line interpreter works in the form of a programming language that can be used for programming and scripting operating system tasks, including the control of peripherals, devices, and system functionality. Shell programming or scripting is the only way to automate one's interactions with an operating system or OS. It allows users to create custom tools and controls to manipulate the OS interactively or non-interactively. While scripting was in use well before, scripting and tools can now be widely used on almost any device.

As part of history, shell scripting has grown with the development of operating systems that coincided with the invention of system commands and functions. From Unix to Berkeley Software Distribution (BSD) and Linux, operating systems were created from smaller operating systems or by altering or adding functions to larger operating systems. Today, shell programming is essential. For instance, all configurations in Unix, PHP, Perl, CGI, and Unix-based Graphical user interface (GUI) programs are run on the device via shell scripting. Even for configuring, managing, or securing Linux systems or networks, one must learn and master shell scripting or shell programming. Editing MACroS (EMAC's) command shell, or simply the EMAC's shell, is a text user interface that emulates a shell on an EMAC's computer. Although it is not a real shell, it allows users to use the text-based interface to navigate menus and run many of the same software programs from its command line, such as the command prompt. Wildcard matching also moves input data into special input file paths, and commands allow for system interactions such as copying input/output. With the glob command package, it can also be used for generating results [3].

What is Shell?

A shell is a user interface for accessing the services of an operating system. It has the following functions. Gather input from the user, files, operating system, command history, etc.–Interpret that input, that is, execute the command that the user gives the shell. Output results of the command back to the user.

Generally, shell allows users to run programs, manipulate files, initiate communication with other programs and systems, and perform other system operations. A shell is as useful as support for these operations. For example, mail clients are great for reading emails, but their utility may be limited by their ability to send attachments, manage large mail directories, and portability or ability to interface with other applications. Operational environments are often a matter of preference [4]. One of the great advantages of Unix and Linux is that they have multiple shells that support an ever-increasing collection of features. This convenience is particularly important when an environment is shared with multiple users. To accomplish more complex tasks effectively and manageably, the shell must be relatively simple and must execute different types of commands. To illustrate, which of these tasks would be simpler for you to perform?

1. Define what happens when a user connects to a server.
2. Make a connection to a server.

The former task likely requires much more labor, or perhaps a third-party manager is needed to complete the work on your behalf. In this way, the shell prevents users from having to learn a lot about system-level programming and the operating system to carry out lower-level tasks [5]. A shell is designed to work from the command line and includes a scripting capability that can automatically alter a computing environment and perform operations on files. Users can customize the computing environment based on their requirements. Shells are very useful for quickly changing a computing environment when a startup depends on the completion of earlier steps. When a user logs into a computer system, that is most frequently a shell running in what is called "interactive mode," responding to commands after a carriage return is pressed. Shells running at this level of operation are often called "command shells." Sometimes, the user does not directly interact with the shell [6]. This often occurs when a shell script or series of shell commands are executed in succession without human intervention. The shell performs the task and does not interact with the user.

Types of Shells

Unix programming environments come with various forms of the included shell, and every operating system can have a different shell as its job control interface. In most cases, no option can be programmed to alter it. Of the larger shells, the differences lie mostly in the syntax and their special features that users appreciate, and at this point, they can choose the right shell to use. There are three types of shells. A flag was used to specify the written shell. For example, it refers to the C shell (csh) and refers to the Bourne shell. The most popular shells were as follows.

1. Bash is a popular alternative to the Bourne shell.
2. Zsh can autocomplete commands while you type them.
3. Tcsh: Turbo C shell is a safer and more powerful version of the C shell.
4. C shell.
5. Fish is the easiest shell to learn.

The Bourne shell (bash) is one of the most robust types of shells. These four simple shell styles are compatible with sh's syntax, as described in this chapter. Compatibility poses a problem because many of the syntax variants that computers produce in binary are often not only in the same language as the shell, but shell implementations cannot correctly predict or diagnose errors in a mixed syntax file. It is dangerous to pass a shell script as input to the shell in an untrusted environment [7]. Data should be collected from an unknown source. Although dated, this bug is still present in modern shells. A better example is that the shift call works differently for tcsh and sh. If I have the following command in a sh script: Shift 10, then shift in a tcsh script. Counting backward in the same script becomes difficult. Simpler factoring can make scripts more portable.

BASIC SHELL COMMANDS

Commands

Essentially, the shell is interactive software that takes commands from the user. Here, we examine some of the basic commands and show how you can use them on their own. For the commands given below, ensure that you type them exactly, as shown here. Unix is case-sensitive, which means that it allows differentiation between upper-case and lower-case letters.

ls

It should be noted that Unix allows all files and directories to be arranged in a structure similar to an organizational chart. Directories can contain files and other directories; thus, a directory can be considered as a list of entries, where each entry consists of two parts: a file name and the complete path to that file [8]. Note that when we refer to "everything on the system," we refer to the file system, not the operating system itself. All "files" on a Unix file system are, directly or indirectly, contained on special files called "partitions" or "filesystems."

ls [OPTION]... [FILE]...

The ls command lists information regarding the files. By default, the ls command lists files and directories in the order in which they occur in the current directory (which is a directory from which you could be working). The options tell the ls command on how to list the content of a directory. Examples of items entering the Unix prompt are as follows:

user@bashbe-1[~] % ls Desktop/Maildir/... user@bashbe-1[~] % ls /etc X11/ aa... xntpd.conf Now try ls with /bin as the argument: user@bashbe-1[~] % ls /bin.

Navigating the File System

In shell programming, it is crucial to understand how to operate with a file system. Three commands that are typically used to navigate and view the file structure from the command line are pwd, cd, and ls. Let us now examine these commands and the options associated with them:

- *The pwd command:* pwd represents the print working directory. It is not a command available in the shell; it is a built-in function.

- *The cd command:* This command is versatile as a developer can switch from a local file system to a remote file system. There are options associated with the CD command. The double dot that follows the CD command is used to move up one level to the parent directory. To move down through the file system, the developer types a / after the cd command and then types the name of the directory that the developer wishes to move into [9]. To move the developer to the home directory of the current user, the developer types after the cd command.
- *ls command:* This command is used to list the contents of the directory.

Navigating Through the Directory Structure

The ability to navigate through directories is necessary for the development of an operating system. As a developer, you will find yourself needing to access files located on the file system of your web server, query the production database to troubleshoot issues, or work on your local machine, Accessing Files-Listing Contents. The Absolute or Relative Path: To manage files efficiently, it is important to understand the difference between relative and absolute paths. Instead of moving around your project with the CD command, you can manually move to any directory from the shell with /, ~, and.. Keep in mind that while one can switch from one user to another from the command line, one will only have the appropriate permission to do so. In most cases, this is not the same as a normal file system. If you are running as a root, you are working in the system root, which can be dangerous. A slight caution when navigating through the file system can save a lot of frustration and headaches. When navigating through Unix/Linux systems, it is necessary to have a good understanding of basic and complex pathname structures, when to use an absolute path, and when to use a relative path to reach certain parts of the directories, locations, and different file system mounts. Specific precautions should always be taken.

File Manipulation

Once you navigate to the directory where you want to work, the next step is to create, delete, or edit files. From the command line, it takes a simple command to edit a file using your favorite editor or to quickly create a file to start. Below, we discuss creating files and directories, copying files, editing files, and deleting them.

Creating Files and Directories

A touch command is used to create an empty file. If an existing file name is not supplied to the touch command, it creates an empty file.

$ touch hello.py $ ls hello.py

The mkdir command is used to create a directory. Like touch, you supply a name to the mkdir command.

$ mkdir data $ ls data/ hello.py

Copying Files

You can copy a file using the cp command along with the source and destination files. This created an identical copy of the original file.

$ cp hello.py good_day.py $ ls data/good_day. py hello.py,

Deleting Files

To delete a file, we use the rm command followed by the file name(s) that you want to delete. $ rm good_day.py $ ls data/hello. py $.

Editing Files

The two command-line text editors were Nano and vi. Nanos are a good choice for beginners. You can simultaneously create and edit files in the text editor by providing the file name as an argument to the editor.

$ nano hello.py $ vi hello.py

Be extremely careful when attempting to edit, delete, or move files in a shell environment. Many of these commands are irreversible and can result in data loss. However, who has not accidentally typed the wrong variable, saved, or exited a file by mistake? This is what this section focuses on. Exercises to display the contents of the files in the command prompt can become good reminders of what you are about to edit, just in case. Here are examples using three different text commands, two of which create new files, whereas the third reverts a file back to its initial state [10].

File manipulation is my favorite thing to do on the command line because it has many real-world applications. Amazingly, you can do this with just a few file manipulation commands. Below are examples of some things that I might do if allowed near the web server.

Use the MV command when you mistyped the name of a file when you created it and want to change it quickly without retyping it in the command prompt. $ mv OldName_20141020.html NewName_NewEvent.html - Use the rm -i command to clean up after yourself. This command asks for confirmation before removing the files so that you do not accidentally delete everything. $ rm -i discrete_modeling.rmd fresh_space.html modeling_part_4.html writing_again.html rm: remove regular file ‘discrete_modeling.rmd’? nrm: remove regular file ‘fresh_space.html’? nrm: remove regular file ‘modeling_part_4.html’? nrm: remove regular file ‘writing_again.html’? n - Use the cp command to back up your writing or website before you make any changes. $ cp writing_again.html writing.html - Use the cp and rm commands together to move files from one directory into another. In this way, you do not have to copy and paste the event pages from one location to live versions.

While editing or deleting files is generally straightforward, it is important to remember that the username is associated with every file created. The changing permissions section explains why this can be a problem. For now, you only need to understand that the owner of a file has exclusive permission to edit or delete the file, and it is very important to be conscientious about creating and deleting files in guest accounts used by other writers or on shared servers.

SHELL SCRIPTING

Shell scripting is writing code to solve your arcane and mundane problems. Whether you need to automate, maintain, or tweak some bit of your system or workflow—invoking long, arcane commands over and over again or tweaking the same one a thousand times, for example—scripting is the answer. Writing even the simplest shell scripts can save hours, and scripts are easy and fun to write. Before continuing with the essay, we need to cover some basic shell workings such as syntax, which will eventually lead us to a discussion of shell scripting. A shell script is written in such a way that it is comprehensible, efficient, and secure to use, and usually would not, given performance considerations, be written in another programming language or concert with another programming language. It was written to run as the shell itself.

A simple shell script can be two or three lines in length. Any command or system function running on the command line can be used in a shell script. The shell script can react to the results of any command as described in the previous paragraph. A shell script accepts parameters from the command line. The commands can be executed from the command line without the need to create elaborate testing harnesses by using other programming languages. For example, let us say that we typed each package file from 1000 necessary CDs to the command line and instead used a quick small shell script to concatenate each file, outputting one file to the screen, and prompting us before printing the next. Therefore, suppose we have a total of 1000 CDs with package files and want to see the contents of File1 for each CD in a series. The following is a shell script that contains one simple shell command: Each command is run in turn and prints the contents of each File1 on the screen.

Variables and Data Types

Variables and data types are the first things one should learn when writing a script in any language, and shell scripting is no exception. Like almost any other programming language, shell scripting allows

the use of variables to store data in memory. The ability to "remember" that such data are stored somewhere allows the script to become more functional. The two most important things you need to know to understand this subject are: A variable can store the values of various data types, including strings, integers, and numerical values, and the data type of variable generally does not matter to the constructor. Based on this requirement, the language automatically changed the data type of variable. There are three fundamental data types in Bourne-Again Shell (BASH) scripting: strings, integers, and arrays. One that we will discuss is the strings.

Strings can be combined with integers such as files, emails, and names. A string may also contain the space between them. To easily understand the BASH string variables, one needs to understand single and double quotes. The variable is treated as a character, variable, special character, or even integer, depending on the type of quote sign used. Some examples of how to declare, assign, and display string variables are as follows:

Name="Aziz belongs to Revopoint" echo $Name
export id=$(echo $BASHPID)
cat iplist.lst

The common problem of setting the environmental variable from beginning to end in the script. See the following script and available solutions: #!/bin/bash export CID=1234 echo "Here: CID=$CID"

Control Structures

In this subsection, we introduce several basic control structures supported by the shell. Control structures are statements that change the execution flow within a program, make decisions, and repeat code execution. These constructions provide us with the ability to branch our code in various directions and perform repetitive tasks that turn command-line usage into efficient real-life automation. After reading this section, one might explore these control structures through practical exercises before moving on to the next section.

In programming, a control structure is generally a statement that directs the code flow. Without control structures, scripts would run from start to finish, following an unchangeable execution path. One of the primary control constructs in shell scripting is the "if" statement, which allows us to make decisions and choose one action from two or more based on a condition. We controlled the exact flow of the script by enclosing the blocks of commands inside pairs of opening and closing statements. In other words, control structures allow code branching within the shell scripting.

Loops are another type of basic control structure that is used in shell scripting. They allow us to repeat code execution, ensuring that a block of commands is executed multiple times according to a predefined number of iterations. These repetitive executions can also be stopped based on specific conditions, such as user input, file content, or the existence of something in the directory, based on commands or their results. Generally, loops provide scripters with a powerful way to automate tasks, turning repetitive tasks into fully automated programs that need to be executed only once for the entire job. In addition to executing commands multiple times, loops can also lead to multiple routes of code, depending on the condition. In several cases, multilevel conditional structures can be constructed by nesting these control structures. Similar to the other control structures, we denote specific loop termination conditions inside the loop body, allowing for potentially infinite iterations. However, they should generally be terminated after a specified upper bound or exit. Error checking concerning the behavior of these feature-rich constructs is addressed in the following sections.

ADVANCE SHELL TECHNIQUES

Some of the following techniques go beyond simple shell scripting but are commonly utilized in advanced scripts. The first is the command substitution. As you advance in scripting and programming,

you will find that one program or script often relies on the output of another, that is, using the output of one command as an argument in another. To illustrate, consider the following commands.

My current working directory is. It would certainly be nice to display the ending directory name without hard coding the script. This can be done with two echo statements, storing the result of the first in a temporary file, and then extracting the information from the temporary file to form the second command. This can be achieved by combining the first two echo statements with a pipe and as an argument to avoid the intermediate file. However, this, in turn, can be avoided by placing parentheses around the command to make the shell execute it first. This is called command substitution. As an example: echo My current working directory is: 'pwd.' The output of the command is automatically passed to the echo argument using the intermediate command. This feature can be used to nest an infinite number of commands. Make it a habit to use command substitutions where feasible - you can save on overhead. The Unix shell provides metacharacters that perform sophisticated pattern matching, called regular expressions. They can be used with any command that allows for pattern matching. Some places where regular expressions can be used are: Looking for a file based on a complex pattern. Use the command. - Want to delete a set of files matching something like "?". Trying commands using regular expressions. - Listing huge files using commands. - Count users in the file used by Unix to keep track of all valid users. - Help search and replace large texts. Care in using regular expressions will make you a great coder in the shell and help you in good pattern matching.

Command Substitution

Command substitution is a feature of the shell that allows you to streamline scripts using the output of one command as the input to another command. When using command substitution, the shell runs the command that you specify between backticks or within parentheses and then replaces the command with its output. This process, known as evaluation, allows one command to be nested within another command. This can be extremely useful and greatly expands the ability to use shell commands and build scripts. Here is a common example of how command substitution is used: the date command, when used with the %Y formatting option, returns to the current year when evaluated. This value is then used by who commands as the first argument, filtering its output. This command displays a listing of all users who are currently logged into the system, along with their respective terminal connections. Any user who logged in during the year and returned by date is listed. A few more examples demonstrate command substitution syntax: 1. user=$(whoami) 2. sum=$((2 + 2)) 3. Total =$(free | grep Mem: | awk '{ print $3 }'). With command substitution, setting the above variables is a two-step process: the relevant command is enclosed in either backticks or the $() syntax, and then the output of the commands is evaluated and placed into the surrounding variable. Command substitution can add a significant amount of clarity to your scripts at times, but one of the major strengths of the feature is its usefulness when processing loops and performing other complex operations. Command substitution is generally quite easy to use, but there are a few things that can go wrong. Perhaps the most common problem is a syntax error. When the backticks or $() sequence that represents command substitution is typed incorrectly—often by omitting or improperly closing the sequence—the executed command will typically fail without providing an obvious explanation. If the output of a command does not behave as expected, it may be useful to confirm that every instance of command substitution has been typed properly. It is also worth noting that the output of a command that relies on command substitution may change unexpectedly if the behavior of any internal command changes. It is always the easiest to debug scripts that do not make use of command substitution, but it can be a very handy technique when used with due care. Exercise command substitution is used to obtain a good feeling about how it works. Keep in mind that because you are manipulating execution order, the different ways you can introduce side effects can sometimes surprise even the most seasoned scriptwriters.

Regular Expressions

Regular expressions are an important feature of shell programming. They allow us to define complex patterns and search within strings of texts. Regular expressions are not specific to shell programming;

they are available in many programming languages and as text editors. In shell programming, they are used for two types of activity. The first is searching within text, such as that performed with the grep command. The second step is to validate the shell code. In this section, we explore how to define patterns using regular expressions.

A regular expression (Regex) is a pattern that can be matched to input text. The input text is searched from start to end to determine whether it matches a regular expression. Here are some fundamental ideas for understanding regular expressions: () – Define a group, [] – Define a character set, – Any character except a newline, ^ – Anchor the start of a line, $ – Anchor the end of a line, \ – Escape the next character. With these operators and special symbols, we can define simple to complex patterns. Some basic rules when designing a regular expression include using $ at the end to prevent partial matching, providing an anchor using ^$ at the start and end for exact strings, and using "\n" in regular expressions as little as possible for efficiency. We must avoid performing regular expressions matching for trivial operations or simple equality checks.

The power of shell programming also comes from the search features. Regular expressions are at the heart of features such as grep. It is also necessary to search for patterns within a text file using this command. In the subsequent exercise, we learn to filter a particular pattern from the sed command. Finally, we use $ to prevent unwanted partial matches. To use regular expressions for optimization, an anchor can be used at the start. Furthermore, we can use the escape symbol to search for any character. Regular expressions have the power to be shell scripts. Knowingly or unknowingly, we use a star to look for a pattern.

BEST PRACTICES IN SHELL PROGRAMMING

This section compiles some best practices in shell programming; the goal is to ensure that the finished script reads clearly and logically without causing the reader to track the state explicitly. The design choices are oriented to maintain the ease of understanding a script at the end of its creation, once it becomes more complex. Scripts are used to build up the major steps necessary for command execution, often involving reading and parsing a configuration file and then optionally preparing a command line or checking for another script's presence before continuing with the main work. By modularizing the process, scripts can be made easier for a new maintainer of the code to understand.

Be liberal with comments and documentation. Always comment on what a script or function is intended to do, and consistently comment on complex codes. Follow the conventions or standards of the language in which you work. Always provide useful feedback to the standard output using a level of verbosity that you would find appropriate when using a new non-interactive command yourself. This issue is linked to error handling. It always provides tools for dealing with error conditions. However, when a tool exists, a wrapper rather than every script should be developed. Develop logging consistency and provide clear error messages at an appropriate level of verbosity for users to engage in effective error handling and debugging. Include an error debug mode, enabled via a command line switch that turns on the verbose output and, in particular, echoes commands as they execute. This makes it easier for the user to trace errors within a script or to understand the detailed work involved in each command when trying to adapt a script for a different system.

Optimizes for performance within reason. This usually means reading files once only, if possible, not running unnecessary code if the result of that code is already known and reducing the code that executes unnecessary subshells. In addition, you always place your most common cases at the top of the case statements. In addition, functions and unnecessary subshells were avoided. Implement safe mode usage, where the script appropriately prints an informative error message and exits early if called without sufficient or correct arguments. The help shall also be printed, but only if the user has asked for it with the correct arguments. Examples include the script being called with an incorrect number of arguments. Plan and write the script thinking about common pitfalls with external and internal commands, as well

as file systems, applications, and user errors. Aim for a script that is robust, at least for common problems, by planning sensibly and defensively, and avoiding being too simple. Simple is beautiful, but it is inaccurate—do something awful, like failing silently. Provide a warning or even an error. More complex scripts are more robust and safer. When writing the script, you may want to look at it in its entirety and all parts to see if it makes sense, then compare the first draft to the final draft of the script. In daily work, an example of the best practice should be to have a set of scripts run to back up the system. When two identical servers exist, each with several terabytes of data, you will create from the most basic to the highest script to generate a dump from all running databases daily. Then, another script compresses the dump, and the last script handles the backup of the data.

CONCLUSION

Shell programming remains one of the most versatile and essential tools in Unix/Linux environments, offering users a powerful means of automating tasks, managing systems, and optimizing workflows. Throughout this paper, *"The Power of the Shell: A Deep Dive into Shell Programming Techniques,"* we explored the foundational concepts and advanced techniques that make shell scripting an invaluable resource for both novice and experienced users. Shell scripts provide a flexible framework for solving complex problems and streamlining daily operations, from control structures and process management to the use of regular expressions and filters.

The simplicity and efficiency of shell programming lie in its ability to bridge the gap between users and the operating system, enabling direct interaction through command-line interfaces. As computing environments continue to evolve, the need for efficient automation and system control has become more critical, and shell scripting remains an indispensable skill for system administrators, developers, and IT professionals.

In conclusion, mastering shell programming empowers users to take full control of their systems, automate routine tasks, and develop solutions tailored to their specific needs. By harnessing the power of shells, individuals and organizations can significantly improve productivity, resource management, and overall system performance, ensuring that shell scripting continues to play a vital role in the future of computing.

REFERENCES

1. Alasmary H, Anwar A, Abusnaina A, Alabduljabbar A, Abuhamad M, Wang A, Nyang D, Awad A, Mohaisen D. SHELLCORE: Automating malicious IoT software detection using shell commands representation. IEEE Internet Things J. 2022;9:2485–96. DOI: 10.1109/JIOT.2021.3086398.
2. Schröder M, Cito J. An empirical investigation of command-line customization. Empir Softw Eng. 2022;27:30. DOI: 10.1007/s10664-021-10036-y.
3. Kidwai A, Arya C, Singh P, Diwakar M, Singh S, Sharma K, Kumar N. A comparative study on shells in Linux: A review. Mater Today Proc. 2021;37:2612–6. DOI: 10.1016/j.matpr.2020.08.508.
4. Stöckle P, Grobauer B, Pretschner A. Automated implementation of windows-related security-configuration guides. Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. New York (NY): Association for Computing Machinery; 2021. p. 598–610. DOI: 10.1145/3324884.3416540.
5. Dai T, Karve A, Koper G, Zeng S. Automatically detecting risky scripts in infrastructure code. Proceedings of the 11th ACM Symposium on Cloud Computing. New York (NY): Association for Computing Machinery; 2020. p. 358–71. DOI: 10.1145/3419111.3421303.
6. Singh SK. Linux Yourself: Concept and Programming. Boca Raton, FL, USA: Chapman & Hall/CRC; 2021. DOI: 10.1201/9780429446047.
7. Švábenský V, Vykopal J, Tovarňák D, Čeleda P. Toolset for collecting shell commands and its application in hands-on cybersecurity training. In: 2021 IEEE Frontiers in Education Conference (FIE). Lincoln (NE): IEEE Press; 2021. p. 1–9. DOI: 10.1109/FIE49875.2021.9637052.

8. Dakic V, Redzepagic J. Linux Command Line and Shell Scripting Techniques: Master Practical Aspects of the Linux Command Line and Then Use It as a Part of the Shell Scripting Process. Birmingham, United Kingdom: Packt Publishing Ltd.; 2022.
9. Rapeli S. Understanding the role of Unix shell in software development and developer experience [Master's thesis]. Espoo (FI): Aalto University; 2024. Available from: https://aaltodoc.aalto.fi/handle/123456789/130272. URN: URN:NBN:fi:aalto-202408255833.
10. O'Neil ST. The command line and filesystem. In: A Primer for Computational Biology. Corvallis (OR): Oregon State University; 2019. Available from: https://open.oregonstate.education/computationalbiology/chapter/the-command-line-and-filesystem/

