Atypon Training Program 2023

OPTIMIZED SHELL SCRIPTING

BY: YUSSIF ABDALLA, The University of Jordan

I. ABSTRACT

This report illustrates the solution of the Atypon's 2023 scripting assignment. The assignment is engineered to provide strong scripting coding skills and it forces participants to review the valuable Linux shell commands. Shell scripting is essential to many programmers who might work as system administrators.

This all-inclusive work in shell scripting comprise design, optimization, and benchmarking of a feature-rich shell script that performs a complex task efficiently. The work focuses on Linux commands, shell scripting techniques, optimizing strategies, and creating a user-friendly experience. Collectively, the work is presented as three distinguished parts: implementation, optimization, and advanced feature integration.

Finally, the work covers design choices, optimization techniques, advanced features, user-friendliness aspects, and their impact on performance as required by the assignment statement. Conclusions, lessons learned, difficulties, and future recommendations are highlighted.

II. INTRODUCTION

Shell scripting is essential to anyone works as a system administrator. Typically, Linux based machines boots up by executing shell scripts in /etc/rc.d to restore the system configuration and set up services. Consequently, a detailed understanding of such startup scripts is important for analyzing the behavior of a system, and possibly modifying it. A shell script may be used to prototype a complex application, where a limited functionality that is generated by a script is often a useful first stage in project development. Shell scripting dates back to the ubiquitous classic UNIX philosophy of breaking complex projects into simpler subtasks, which may be used for chaining together components and utilities [1][2].

In this work, a report summary of the design, optimization, and benchmarking of a feature-rich shell script that performs a complex task efficiently is presented. The work focuses on Linux commands, shell scripting techniques, optimizing on strategies, and creating a user-friendly experience. Collectively, the work is presented as three distinguished parts: implementation, optimization, and advanced feature integration [3].

Finally, the report covers the design choices, optimization techniques, advanced features, user-friendliness aspects, and their impact on performance. It also include a reflection section discussing the lessons learned, difficulties encountered, and potential future improvements.

III. SCRIPT DESIGN AND IMPLEMENTATION

In this first part, complete overview of script design and its implementation is presented, highlighting the most essential aspects of the design.

3.1 Script Flow chart

A flowchart of the proposed program may be found as depicted in Figure (1), where options are highlighted in green text.

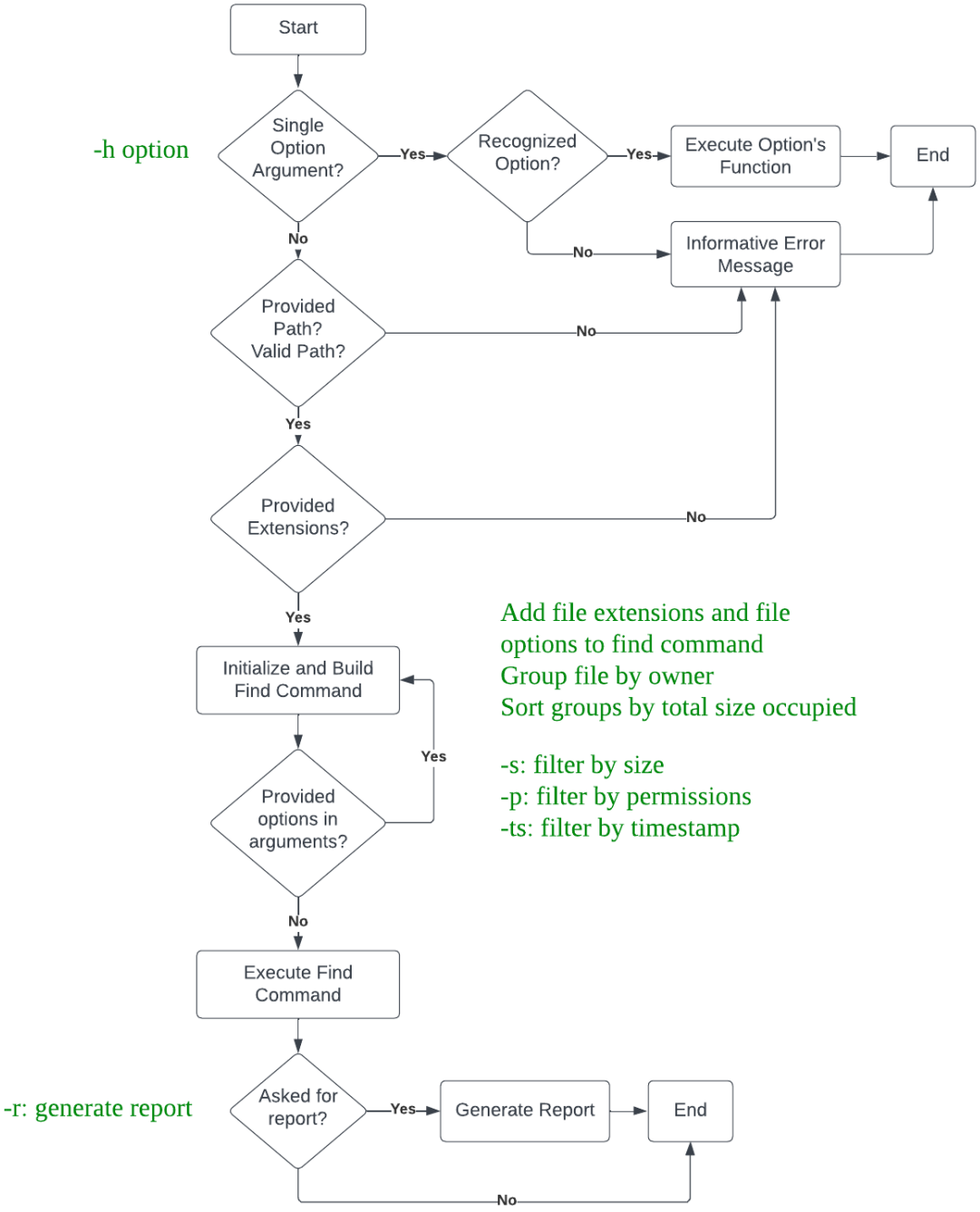

Figure 1: The Proposed Script Flow Chart

3.2 Usage and Syntax

The script begins by processing the arguments, which can be provided in two different command formats. In the first command format, the user only provides a single option. These single options are the ones that do not require any additional arguments to function.

For example, the "-h" option provides information on built-in commands, explaining their purpose and usage.

```
bash script.sh -h
```

In this first format style, the script checks for single arguments and invokes a dedicated function for each option. Appendix A provides a detailed overview of the script's structure and how it handles single arguments. Please refer to Appendix A for a comprehensive presentation of how the script is coded and how it is effectively handles these arguments. For instance, the "-h" option specifically invokes the Help() function, which displays valuable information on the screen.

In the second format style, the user provides a minimum of two arguments. The first argument must be the directory path in which the command will search, followed by one or more file extensions that the user wants to look for within the specified directory path.

```
bash script.h <directory path> <ext1> <ext2> ...
```

3.3 Initializing the Find Command

The script creates an array of file extensions provided by the user. It then constructs a find command to search for files with the specified extensions in the given directory and its subdirectories. The find command is set to search for files (-type f) in the specified directory.

Now, to filter files based on the specified extensions, the script iterates over the array of extensions. For each extension, it appends the -name option with the pattern "*.<extension>" to the find command. This pattern enable matches files with the current extension. The script uses the logical OR operator (-o) to separate each extension. This means that the find command will match files with any of the specified extensions.

Finally, the script appends -false to the find command to ensure that the preceding conditions are grouped correctly as a single logical unit. When executed, the find command will search for files in the given directory and its subdirectories, filtering them based on the specified extensions. The resulting files will be used for further analysis or processing. Please refer to Appendix B for the complete code listing and further details.

3.4 Building the Find Command

An associative array is used to store key-value pairs representing the available options for filtering files. The keys correspond to the options that users can use to filter files, such as **-s** for size, **-p** for permissions, and **-ts** for timestamp. The values associated with these keys are the corresponding options used by the find command to apply the filters, namely **-size**, **-perm**, and **-newermt**.

The purpose of this array is to dynamically construct the find command based on the user's requested options. For instance, if the user includes the -s option to filter files by size, the script retrieves the corresponding option **-size** from the array and adds it to the find command. Similarly, if the user specifies the **-p** option for filtering by permissions, the script appends the **-perm** option to the find command. Likewise, the **-ts** option for filtering by timestamp results in the addition of the **-newermt** option to the find command.

By utilizing this associative array, the script can dynamically generate the appropriate find command based on the user's selected options, allowing for customized file filtering based on size, permissions, or timestamp. This is achieved by iterating through the command-line arguments and checking if an option provided by the user is defined in the associative array. If a matching option is found, the script looks for the corresponding option parameter in the next argument. Once the option and its parameter are obtained, they are added to the find command to apply the desired filter. This iterative process ensures that the script accurately constructs the find command with the user's specified options and parameters.

After processing the filter options, a check for the presence of the -r option is conducted, which enables the user to generate a report. This option functions slightly differently compared to the others. It does not require a corresponding parameter and does not directly affect the find command. Instead, when the -r option is detected in the arguments, the bode sets the **r_option_found** flag to be true. This flag serves as an indicator, allowing us to identify if the user specified the **-r** argument. One can then process this option later, separately from the find command, to generate the desired report. Please refer to Appendix C for the complete code listing and further details.

3.5 Finalizing and Executing the Find Command

After the script has finished processing the user's options, the **ls -lh** command is added to the find command. This command is responsible for displaying detailed information about the files, including their size, owner, and permissions.

Following that, the **awk** command is executed to process the output of **ls -lh**. It performs several tasks: grouping the files by owner, calculating the total size for each owner, and printing the results. This step allows for the files to be categorized according to their respective owners.

Finally, the **sort** command is utilized to sort the file groups based on the total size occupied by each owner. By specifying the appropriate sorting options, the files are arranged in descending order of their total size, ensuring that the largest file groups are displayed first.

Once we have constructed the complete find command, it is executed by using the **eval** command and storing the results in the **file_analysis.txt** file. For a detailed code implementation and further information, please refer to Appendix D.

3.6 Generating a Report

In this part, the script checks if the **r_option_flag** is set to true. If so, the script will generate a summary report that displays the total file count, total size, and other relevant statistics. The file count is found by piping the **find_command** command to the **wc -l** command, which counts the number of lines in the output.

Furthermore, the total size is found by piping the **find_command** command to the awk command, which is used to calculate the total size of files. One may refer to Appendix E for the complete code listing and further details.

IV. USED OPTIMIZATION TECHNIQUES

In this second part, the used optimization techniques in this work are highlighted. Two namely used techniques are presented; single find command usage and the use of Use of Associative Arrays.

4.1 Using a single find command

A key optimization technique that is used in the presented script is the utilization of a single find command. Any options provided by the user are mapped to corresponding parameters appended to the find command. By processing the user's options, a single find command is employed to filter and locate all file extensions within a directory or subdirectory. In contrast, a less optimized script would employ a distinct find command for each extension or filter.

Utilizing a single find command provides a significant optimization benefit by avoiding redundant directory traversals and unnecessary overhead. In a less optimized approach, each find command would traverse the directory structure separately for each extension, resulting in inefficiencies and unoptimized steps. However, by consolidating multiple file extensions into a single find command, the script only needs to traverse the directory structure once, leading to enhanced performances and improved efficiencies.

Another advantage of employing a single find command is the reduction in required system calls for initiating and managing each command. Since system calls have an associated overhead, minimizing their number enhances the overall performance of the script and improves efficiencies.

Moreover, using a single find command simplifies the code and reduces complexity. It makes the code more concise, readable, and easier to understand and maintain.

In summary, employing a single find command for all file extensions offers substantial optimization benefits by minimizing redundant traversals, reducing system calls, and consequently, simplifying the code.

4.2 The Use of Associative Arrays

Associative arrays are considered as an optimization technique in shell scripting as they contribute to improved efficiencies. These arrays enable efficient data lookup by allowing to store and retrieve values using keys. This eliminates the need for complex loops and it provides a straightforward way to access the data.

In the presented script, the use of an associative option array replaces the need for multiple conditional statements. With associative arrays, there is no longer a requirement for repetitive code blocks. Instead, one can access data using keys, eliminating the need for duplicating similar code. This optimization technique enhances the code by making it more concise and manageable.

Additionally, associative arrays enhance code readability by providing a structured approach to store and access data. These arrays also improve code maintainability by facilitating easy updates to the data structure.

In summary, utilizing associative arrays as optimization techniques in shell scripting results in improved efficiency, streamlined code, enhanced readability, and simplified maintenance. The code in Figure (2a and 2b) contrasts the use of associative arrays.

```
args=("$@")
for ((i = 0; i < ${#args[@]}; i++)); do
  arg="${args[i]}"

  if [[ "$arg" == "-s" ]]; then
    ((i++))
    s_option_parameter="${args[i]}"
    find_command+=" -size $s_option_parameter "
    continue
  fi

  if [[ "$arg" == "-p" ]]; then
    ((i++))
    p_option_parameter="${args[i]}"
    find_command+=" -perm $p_option_parameter "
    continue
  fi

  if [[ "$arg" == "-ts" ]]; then
    ((i++))
    ts_option_parameter="${args[i]}"
    find_command+=" -newermt $ts_option_parameter "
    continue
  fi

  if [[ "$arg" == "-r" ]]; then
    r_option_found=true
    continue
  fi

done
```

Figure 2.a Code without associative arrays

```
declare -A options=(
  ["-s"]="-size"
  ["-p"]="-perm"
  ["-ts"]="-newermt"
)

args=("$@")
for ((i = 0; i < ${#args[@]}; i++)); do
  arg="${args[i]}"

  if [[ "${options["$arg"]+defined}" = "defined" ]]; then
    option="${options["$arg"]}"
    ((i++))
    value="${args[i]}"
    find_command+=" $option $value"
  fi

  if [[ "$arg" == "-r" ]]; then
    r_option_found=true
  fi

done
```

Figure 2.b Code using associative arrays

4.1 Using wc -l command and avoid storing to file

When generating the summary report and determining the file count, the script executes the find command and passes its output to wc -l. In a previous approach, I stored the output in a text file before counting the lines. However, the current approach offers the following optimizations:

Reduced memory usage: By directly using the find command with wc -l, the script avoids storing the entire output in memory or on disk. This optimization prevents excessive memory consumption and allows for efficient line counting without requiring substantial memory resources.

Minimized processing overhead: The use of wc -l for line counting is a lightweight operation. It doesn't involve additional processing steps, resulting in reduced processing overhead. By eliminating the intermediate step of storing the output in a file, the script executes the line counting operation more efficiently and with improved performance.

These optimizations collectively enhance the script's efficiency by minimizing memory usage and reducing processing overhead.

V. USER FRIENDLINESS ASPECTS

The script is designed to provide a user-friendly experience by incorporating the following several features:

- **Clear Prompts**: The script includes clear prompts throughout the execution, guiding the user through the process. For example, when the user forgets to provide a directory path or file extension, the script kindly reminds them to provide the required information.
- **Error Handling**: The script performs error handling to ensure smooth execution. It checks if the provided directory path is valid and exists. If the path is invalid or not found, the user is notified of the issue, allowing them to rectify the problem. Similarly, if the user forgets to enter a file extension, the script prompts them to provide at least one file extension.
- **Help Section**: The script provides a helpful option for users to request assistance. By using the command bash script.sh -h, the script prints a brief explanation of its functionality, how to use it, and the available command options. This help section serves as a handy reference for users, allowing them to understand and utilize the script effectively.
- **Examples and Usage**: Along with the help section, the script also provides examples of different command options and usages. These examples further enhance the user's understanding of how to utilize the script for their specific requirements. By demonstrating various usage scenarios, the script empowers users to make the most of its capabilities.

By incorporating these features, the script aims to create a user-friendly experience that is both informative and intuitive. It guides users through the process, handles errors gracefully, and provides clear instructions and examples for effective usage.

For insights into how the script delivers a user-friendly experience and handles errors, please refer to Appendix F.

VI. CONCLUSIONS, REFLECTIONS AND FUTURE WORK

Complete design, optimization, and benchmarking of a feature-rich shell script that performs a complex task efficiently is successfully coded and presented. The work mainly focuses on Linux commands, shell scripting techniques, optimizing on strategies, and creating a user-friendly experience. Collectively, the work is presented in detailed as three distinguished parts: implementation, optimization, and advanced feature integration.

6.1 Lessons Learned

I have gained proficiency in various Linux commands and their practical usage. Through this, I have learned how to effectively combine Linux commands to automate

repetitive tasks and streamline workflows. Additionally, I have acquired the ability to automate tasks according to specific requirements and customize system settings accordingly.

The command line interface has complimented my knowledge, and it allowed me to confidently navigate and manipulate the system commands. This command interface familiarity has significantly increased my efficiency, learning curve and productivity.

Moreover, I have developed strong problem-solving skills, enabling me to break down complex problems into manageable tasks and produce the necessary solutions to such problems. This assignment enabled and sharpened my skills and abilities in logical thinking, troubleshooting, and debugging.

The ability to understand the needs of users and facilitating their search for information have been key focuses for me. By providing clear prompts and implementing effective error handling, I was able to create a user-friendly experience that simplifies the user interactions with the system.

I have realized that Linux and shell scripting are continually evolving technologies. As I delved into these fields, I realized the importance of staying updated with their latest advancements. New tools, methodologies, and scripting languages emerge regularly, making continuous learning and improvement a necessity.

6.2 Difficulties Encountered

During my journey, I have encountered two significant difficulties. Firstly, I faced challenges due to my lack of familiarity with the wide range of options and commands available in Linux! I often found myself searching for specific commands that would suit my design requirements. This process required extra effort and time to find the most suitable solutions. Secondly, I struggled with understanding how different commands interacted with each other. As I combined commands to achieve desired outcomes, it took some trial and error to grasp their interactions and ensure they worked together seamlessly. However, through persistence and continuous learning, I was able to overcome these difficulties and expand my knowledge and proficiency in Linux and shell scripting.

6.2 Potential Future Improvement

I believe there is room for improvement in the script by introducing additional filtering options. For instance, the ability to filter files based on their timestamp, such as filtering by the last X days, minutes, or hours, would enhance its functionality.

Also, providing various export formats, including CSV or JSON, would offer users more flexibility in handling and analyzing the generated reports.

Another valuable addition would be the option for users to decide whether they want to search for the specified extensions in the directory alone or include subdirectories as well. This feature would enable a more comprehensive analysis based on the user's requirements.

Finally, enhancing the generated report by incorporating additional relevant statistics would provide users with a more comprehensive overview of the analyzed files. This could include statistics such as file count by owner, file size distribution, or file permissions summary…etc.

Hopefully, by implementing these improvements, the script can offer users enhanced functionality, increased flexibility, and more informative analysis reports.

V. REFERENCES

[1] Mendel Cooper, An in-depth exploration of the art of shell scripting, PUBLICDOMAIN 10 Mar 2014, Revision 10.

[2] Chet Ramey, Bash Reference Manual, Case Western Reserve University Brian Fox, Free Software Foundation, Sep. 2022.

[3] Dr. Motasem Aldiab , Shell Scripting Assignment, Atypon Training Program 2023.

VI. APPENDICES

Appendix A

```
Help()
{
  echo "This script performs comprehensive file analysis in a given direc-
tory."
  echo "It searches for all files with a specific extension in the given di-
rectory and its subdirectories,"
  echo "generates a comprehensive report including file details like size,
owner, permissions, and last modified timestamp,"
  echo "groups the files by owner, sorts the file groups by the total size
occupied by each owner,"
  echo "and saves the report in a file named 'file_analysis.txt'."
  echo ""
  echo "Users of this script can filter files based on size, permissions, or
last modified timestamp,"
  echo "enabling customized search criteria, and have the ability to generate
a summary report"
  echo "that displays total file count, total size, and other relevant sta-
tistics."
  echo ""
  echo "Usage:"
  echo ""
  echo "  $0 <option>"
  echo "  $0 <directory path> <extensions> [options]"
  echo ""
  echo "Options:"
  echo ""
  echo "  -h      Display this help message"
  echo "  -s      Filter based on the size of the files"
  echo "  -p      Filter based on the permissions"
  echo "  -ts     Filter based on the time stamp"
  echo "  -r      Generate a summary report that displays total file count,
total size, and other relevant statistics"
  echo ""
  echo "Example usage:"
  echo ""
  echo "  $0 -h"
  echo "  $0 <directory path> <extensions> -s <size>"
  echo "  $0 <directory path> <extensions> -p <permission>"
  echo "  $0 <directory path> <extensions> -ts <time stamp>"
  echo "  $0 <directory path> <extensions> -r"
  echo ""
}

while getopts ":h" option; do
        echo $option
   case $option in
      h) Help
         exit;;

      \?) echo "Error: Invalid option"
         exit;;
   esac
done
```

Appendix B

```
initialize_find() {
  local directory="$1"
  shift
  local extensions=("$@")

  local find_command="find \"$directory\" -type f \( "
  for extension in "${extensions[@]}"; do
    find_command+="-name \"*.$extension\" -o "
  done
  find_command+=" -false \)"

  echo "$find_command"
}
```

Appendix C

```
declare -A options=(
  ["-s"]="-size"
  ["-p"]="-perm"
  ["-ts"]="-newermt"
)

args=("$@")
for ((i = 0; i < ${#args[@]}; i++)); do
  arg="${args[i]}"

  if [[ "${options["$arg"]+defined}" = "defined" ]]; then
    option="${options["$arg"]}"
    ((i++))
    value="${args[i]}"
    find_command+=" $option $value"
  fi

  if [[ "$arg" == "-r" ]]; then
    r_option_found=true
  fi

done
```

Appendix D

```
# Finalize the find command
find_command+=" -exec ls -lh {} \; | awk '{ sum[\$3] += \$5; print } END {
for (owner in sum) print \"Total size for\", owner, \":\", sum[owner] }' |
sort -k6,6nr "

# Run the find command and save output to file_analysis.txt
eval "$find_command" > file_analysis.txt
```

Appendix E

```
if [[ "$r_option_found" = true ]]; then
  find_count=$(eval "$find_command | wc -l")
  echo "Total count: $find_count"

  files=$(eval "$find_command")
  total_size=$(echo "$files" | awk '{sum += $5} END {print sum}')

  average_size=$((total_size/find_count))
  echo "Average size: $average_size"
fi
```

Appendix F

```
if [ -z "$1" ]; then
  echo "Please provide a directory path."
  exit 1
fi

# Check if the directory exists
if [ ! -d "$1" ]; then
  echo "Directory not found: $1"
  exit 1
fi

# Check if at least one file extension is provided
if [ $# -lt 2 ]; then
  echo "Please provide at least one file extension."
  exit 1
fi
```