Degree project

Comparison of POSIX-compliance and performance of Linux shells

Bachelor Degree Project in Informatics

First Cycle 30 credits

Spring term 2025

Student: Pontus Lahtinen (a22ponla@student.his.se)

Supervisor: Johan Zaxmy

Examiner: Thomas Fischer

ABSTRACT

Shells are an important part of most operating systems, acting as the part that allows the user to communicate with and send commands to the operating system itself. Many shells exist, especially in Linux environments, but not much research has been performed on them.

This work looks at nine different Linux shells, Bash, dash, yash, fish, ksh, mksh, zsh, tcsh, and OSH, and compares them on how compliant they are with POSIX (portable operating system interface) and performance. To achieve this goal different tests were presented and used to quantify both POSIX-compliance and performance. POSIX-compliance was tested by running different test suites aimed at testing POSIX functions and helping with scripting compatibility. Performance was tested by setting up different scripts, in part with the help of the *Shellbench* program, that tested how fast the different shells could run different functions/commands.

The results found that in terms of POSIX compliance, the shells that aim for POSIX compatibility all have equal levels of compliance, with Bash being the most compliant. In terms of performance the shell that stood out the most was dash, performing the best on every test except for the tests where subshell functionality was tested, where ksh was the fastest due to a unique optimisation involving subshells.

Keywords: POSIX, shell, Linux, performance, scripting

Contents

1 Introduction....1
1.1 Aims....1
1.2 Report outline....1
2 Background....2
2.1 Preliminaries / Concepts....2
2.1.1 POSIX....2
2.1.2 Computer Shells....2
2.1.3 Shell features defined in POSIX....5
2.2 Research background....6
3 Problem formulation....7
3.1 Study aims....7
3.2 Motivation....7
3.3 Delimitations....7
4 Methodology....8
4.1 Data collection....8
4.2 Data analysis....9
4.3 Validity and Reliability....9
4.3.1 Conclusion Validity....9
4.3.2 Internal Validity....10
4.3.3 Construct Validity....10
4.3.4 External Validity....10
5 Implementation....11
5.1 Environment....11
5.2 POSIX Tests....12
5.3 Performance tests....13
5.3.1 Shellbench....13
5.3.2 Handmade Scripts....14
6 Results....17
6.1 POSIX Tests....17
6.2 Performance tests....19
7 Discussion....25
7.1 In relation to previous research....25

7.2 Ethical and societal aspects....28
8 Conclusion....29
8.1 Future work....29
References....30
Appendices....32
Appendix A – Failed Smoosh Tests....32
Appendix B – Modernish Quirks and Bugs....38

1 Introduction

The shell is an important part of every modern operating system (Pickle, 2023). It is the part of the system that the user interacts with to launch programs, view and manipulate the file system, interact with running processes, and so on. In the context of UNIX-like operating systems, there are many different available shells for users to choose from. Most of these shells try to follow the Portable Operating System Interface (POSIX) standard (most recently POSIX.1-2024), to ensure portability with scripts between different shells, as well as for familiarity for users. This thesis work looks at many different shells available on Linux and compares them to each other, to see to which extent they actually follow POSIX, and to see how well they hold up in terms of performance.

1.1 Aims

The aim of this thesis work is to compare different Linux CLI shells when it comes to performance and POSIX-compliance.

1.2 Report outline

In chapter two, the background to the work will be introduced, explaining concepts such as what a shell and POSIX is and what previous work has been done in the area. In chapter three the aim of the thesis will be explained more thoroughly as well as motivated. In chapter four the methodology will be explained and motivated. In chapter five the implementation will be explained, showing the environment and how the tests were performed and set up. In chapter six the results of the tests will be presented, and the results will then be discussed and analysed in chapter seven. Finally, in chapter eight the entire work will be summarised and concluded, along with some ideas for future work that can be done in the area.

2 Background

This chapter provides a background to concepts that are central to this study as well as previous research in the area.

2.1 Preliminaries / Concepts

The following section explains concepts important to the study.

2.1.1 POSIX

In 1980 an organisation made up of commercial UNIX users named "/usr/group" was created. The organisation was created as a response to many different "UNIX-like" operating systems being developed at the time. UNIX-like operating systems are operating systems that were made to work and act like UNIX, but were not necessarily "real" UNIX, i.e. not licensed by AT&T (then developers of UNIX) and did not strictly copy UNIX code. Examples of different UNIX-like operating systems that were on the market at the time were: AT&T's official "Unix System 3", the University of California, Berkley's "BSD version 4", Whitesmith's "Idris", Mark Williams Company's "Coherent", and Charles River Data Systems' "Unos". /usr/group's goal was therefore to create a standard for operating system interfaces that was "vendor-independent". In 1985 /usr/group's work to create a standard moved to IEEE, who earlier also had started a project to create a operating system standard based on UNIX. At first the work was only focused on program interfaces, but eventually also started encompassing a lot of different areas of operating systems as a whole, including the shell (Isaak, 1990). The standard definitions for the shell was chosen to have the UNIX System V shell as the starting point, which is a version of the Bourne shell (sh) (The IEEE and The Open Group, 2024).

The name "sh" is nowadays still often used to invoke a fully POSIX-compliant shell, the Debian Policy Manual for example specifies that "Scripts may assume that `/bin/sh` implements the POSIX.1-2017 Shell Command Language" (Debian Project, 2025), and Bash is launched in POSIX mode when invoked with the "sh" name, which removes all "bashisms", i.e. unique Bash features (Free Software Foundation, 2022).

2.1.2 Computer Shells

The "shell" in a computer operating system (OS) is an integral part and serves as the part that allows the user to communicate with the actual operating system and its processes. The modern shell is used for opening files, browsing folders, running and terminating applications, viewing processes, and so on. Shells can be both a graphical user interface (GUI) and a command-line interface (CLI), and most operating systems have both at the same time. Windows, for example, has the GUI shell "explorer" and the CLI shell "cmd". When it comes to UNIX-like operating systems, there are several CLI shells available for users to choose between, such as the popular "Bourne-Again Shell" (Bash), "Bourne shell" (sh), and "Z shell" (zsh). CLI shells are also script programming languages, allowing the user to set up scripts to perform predetermined tasks that the shell can do, useful for automating repetitive tasks (Pickle, 2023).

The first UNIX shell was made by Ken Thompson and released in 1971. This shell introduced some features that are still very common today in shells, like piping with "|", which makes output of one

command become the input of the next, and redirection symbols like ">" which redirects the output of the command to a chosen location/file. The Thompson shell was however very simple, with barely any support for scripting, instead acting mostly as an interactive shell (i.e. users typing commands into a command line). A new shell was soon written by Steve Bourne, the so-called "Bourne shell" (sh), which released with UNIX version 7. Bourne's goal was to make the shell a full programming language to be used for scripting, along with also being an interactive shell. To this end, the Bourne shell introduced control flow functions such as "`if`" statements and "`for`" loops, as well as variables and command substitution (Computerwoche, 2009; Jones, 2011).

Around the same time the Bourne shell was being developed, the "C shell" (csh) was being developed by Bill Joy for the Berkeley Software Distribution (BSD), a UNIX-like operating system. The goal of the C shell was to be a shell with programming that had similar syntax to the C programming language. Some useful features introduced in csh were command history and job control. The Tenex C Shell (tcsh) was later developed by Ken Greer, which is fully compatible with the C shell, but brings some new functionally that was available on the TENEX operating system at the time, such as command completion (pressing a button, usually tab, to auto-complete a partially written command or file path) and command-line editing (Jones, 2011).

The KornShell (ksh) was a shell created around the same time as tcsh by David Korn, it had features from the, at the time modern, C shell such as command history, but was backward-compatible with the Bourne shell (sh). It also introduced new scripting functions that made it stand out, such as float point calculations and associative arrays (Jones, 2011).

The Bourne-Again Shell (Bash) was later created by Brian Fox for the GNU Project to serve as a free and open source replacement for the Bourne shell, being backwards-compatible with scripts from sh, while also introducing many new features that were available on other modern shells at the time (ksh and csh), like command history, job control, and command completion. Bash has now become one of the most popular shell, being available and the default for many UNIX-like operating systems (Computerwoche, 2009; Jones, 2011).

The "Z shell", or zsh, is a shell created in the 1990's by Paul Faulstad and was designed to be a highly customisable interactive shell. It is based on ksh and incorporated features from csh and tcsh such as command history, command completion, and globbing (auto completing filenames) (Machado, 2024; Z Shell, 2010). zsh has also been the default on macOS since macOS Catalina (Apple, 2023).

Another Linux shell is **not** POSIX-compliant, but still relatively common (common for being non-POSIX) is fish (friendly interactive shell). Its core design principle is to be first and foremost user friendly and intuitive, and only follows POSIX standard where the developers feel that the POSIX standard does not conflict with those design principles (Fish Shell, n.d.-a). For example, command substitution can not be done with backticks as the POSIX standard defines: `` `command` ``, as backticks can be easy to confuse and misinterpret, especially if there is a lot of quoting going on in the same line. Fish does support command substitution with dollar and parentheses: `$(command)`, which is POSIX-compliant, but fish also has its own way of command substitution with just parentheses: (command), which is not defined in POSIX (Fish Shell, n.d.-b).

The "Almquist Shell" (ash) is a shell written by Kenneth Almquist as a reimplementation of the UNIX System V shell (sh), released in 1989. It was adopted as the default shell in BSD-Net/2 in 1991, and kept getting updated with new BSD releases over the years. In 1997 the ash version that was available in NetBSD was ported to the Linux distribution Debian by Herbert Xu which became known as

“Debian Ash” (dash). Goals of dash are to be completely POSIX-compliant, as well as being small in size for efficiency (Maschek, 2021). In 2006 Ubuntu made dash the default system shell (i.e. the shell used when running scripts and that is used when booting Ubuntu, the shell that is used when a user opens a terminal and writes in it (the interactive shell) is still Bash). The reason for this is the better performance, specifically during the startup of the operating system (Ubuntu Wiki, 2017).

“Yet another shell” or yash, is a shell written in C by Yuki Watanabe that aims to be the most POSIX-compliant shell in the world. It is currently compliant to the latest POSIX version POSIX.1-2024, with some exceptions listed on the GitHub where it is non-compliant (Watanabe, 2025).

The “MirBSD Korn Shell”, or mksh, is a derivative of pdksh (Public Domain Korn Shell) which in turn is a derivative of the original KornShell. It is made for the MirBSD operating system but is also available on other UNIX-like operating systems, such as Linux (MirBSD, n.d.).

Oils is a project made up of a shell interpreter that runs in two “modes”, functionally acting as two different shells, OSH and YSH. OSH aims to be a POSIX-compliant shell, or at least a shell that can run POSIX-compliant scripts as well as Bash scripts. YSH is designed to be a new shell language, more similar to Python and JavaScript. Oils is meant to serve as a way to “evolve” the shell from the POSIX standard/Bash, by giving a way to easily go from old scripts that can run in OSH to the new YSH language (Oilshell, 2023).

Table 1 shows all of the shells that were used in this study along with general information about them:

Shell	Language	Licence	Website	First Release
Bash	C	GPLv3	gnu.org	1989
dash	C	3-clause BSD	apana.org.au	1997
yash	C99 (ISO/IEC 9899:1999)	GPLv2	magicant.github.io	2007
fish	Rust	GPLv2	fishshell.com	2005
ksh	C	EPL-2.0	kornshell.com github.com	1983
mksh	C	MirOS Licence	mirbsd.de	2006
tcsh	C	3-clause BSD	tcsh.org	1983
zsh	C	MIT	zsh.org	1995
OSH	C++	Apache-2.0	oils.pub	2017

Table 1: Table of Shells

2.1.3 Shell features defined in POSIX

POSIX defines many things when it comes to the operating system, but most relevant for this thesis work are the parts defining the shell. Notably, the shell itself is defined as a "command language interpreter" based off sh. This definition therefore automatically excludes GUI shells, as POSIX does not see these as shells. Some examples of what POSIX defines in regards to the shell is redirection with ">", piping with "|", commands like "`if`", "`elif`", "`then`", and "`do`" for use in scripting, how variables should work, command history, basic commands like "`cd`" to change directory and "`login`" to login as a user, using backslash to "escape" characters (type out the actual character after the backslash instead of performing the function the symbol normally would, for example a curly bracket "{") (The IEEE and The Open Group, 2024).

Job control is the function of being able to start, stop, pause, and continue "jobs"/processes within the shell. A running process/job inside the shell can be suspended by sending a SIGSTOP signal to the process, usually done by pressing CTRL+Z or by using the "`kill`" command (which is used to send signals), the suspended job can then be continued in the foreground or background by using the "`fg`" or "`bg`" commands respectively, or by sending the SIGCONT signal to the job. The "`jobs`" command can be used to get a list of current jobs along with number identifier (The IEEE and The Open Group, 2024).

Command substitution is the feature of substituting the output of a command in place of the command, for example if one wanted to make a folder with the name of the current year and month, the "`date`" command (also a POSIX defined command) could be used in command substitution: "`mkdir $(date +%Y-%m)`". Command substitution can be done either with backticks `` `command` `` or dollar and parentheses `$(command)` (The IEEE and The Open Group, 2024).

Command-line editing refers to the feature of being able to edit commands while typing them in a terminal (for example moving the cursor left or right with the arrow keys). POSIX defines that the command "`set -o vi`" should set command-line editing to vi-mode (allowing for other editing modes to be default). When in vi-mode editing the command-line should act as in the vi text editor, where, for example, the "h" key can be used to move the cursor left and "l" to move the cursor right, CTRL+W can be used to delete the last word, pressing the "kill" character (set by the "stty" program) removes the current line and pressing the "erase" character (also set by "stty") removes the current letter the cursor is on and goes to the left once (The IEEE and The Open Group, 2024).

"Built-In Utilities" are utilities and programs that are implemented into the shell itself instead of having to access an external program for it. These are utilities that are or historically were, commonly used. One example is the "`time`" command, which measures the time it takes to run a command in real time, as well as how long the CPU spent time running the command in userspace and how long it ran in kernel space. Some other examples are the "`true`" command, which simply returns a true value successfully, the "`touch`" command which updates the last accessed and last modified metadata (also commonly used to create new empty files), and the "`uname`" command which returns the name of system (e.g. "Linux"). Most built-ins are mandatory by POSIX, but some uncommon ones are optional.

Subshells are child processes created by the shell in certain circumstances that inherits the environment from the parent shell. For example during command substitution, the command inside the parentheses (or backticks) are run inside a new subshell that the shell created, which then returns the output of the command to the parent shell. Piping commands also creates a new subshell for each new pipe.

2.2 Research background

Kidwai et al. (2021) look at seven of the most popular Linux shells and compares them in a few areas, for example, security features (such as if the shell has a special prompt for entering passwords securely), interactive features (such as command auto-completion), programming features (such as handling float point arithmetic), and inter-process communication (such as command substitution). In the conclusion they mention the Z shell as being the most outstanding Linux shell, at least when it comes to the factors that were looked at in their study.

In Greenberg & Blatt (2019), the authors describe "Smoosh", which is a formalisation of the POSIX shell, intended to make it easier to perform research on the POSIX shell. When arguing for if Smoosh is actually a POSIX-compliant model, they mention (and show) three different test suites that they used to test POSIX-compliance. One is the official test suite distributed by the Open Group for POSIX certification[1], another suite is using tests from the "Modernish" library, which is a library meant to help write POSIX-compliant shell scripts, included in this library is a suite of tests meant to ensure portability for scripts. Finally, the authors created a test suite of their own to test POSIX-compliance. The authors run these test suites on Smoosh as well as on seven different common Linux shells: Bash, dash, zsh, mksh, ksh, yash, and OSH. Here the different shells show different results even among each other in all the test suites, which points to there being different levels of POSIX-compliance in different shells that all aim to be POSIX compliant.

Greenberg et al. (2021) talk about the UNIX shell's future. They mention what is good about the inherent characteristics of the shell, as well as what is bad, and how these characteristics can make it troublesome to research and improve the shell. They also talk about recent developments that help alleviate some of these problems, which makes it easier to further research, most relevant of these is "Smoosh", a formalisation of the POSIX shell standard. They also mention how the UNIX shell is an under-researched area when it comes to computers in general.

No other scientific research reports or similar could be found on the topic of Linux shell comparison, most material is scripting tutorials or guides on how to use the "UNIX shell" in general. There are some non-scientific material that compare different shells, such as the Wikipedia page on "Comparison of command shells" (2025) which lists a lot of different shells and characteristics of them, such a license, release date, interactive features, etc., however not in terms of performance or POSIX-compliance (though some of the features listed are defined in POSIX).

1 https://posix.opengroup.org/testsuites.html

3 Problem formulation

In academic research, the topic of shells is a relatively under-researched area compared to other areas of computers, and not much work has been done on it. One of the papers that do exist on the topic of shells, that was looked at in the research background (Greenberg et al., 2021) also mentions how the UNIX shell in general is a pretty under-researched area. One of the articles that does look at different shells (Kidwai et al., 2021) and compares them, does not compare the shells when it comes to performance and POSIX-compliance, which is what this thesis work will focus on, and the amount of shells compared is only seven, which is a relatively small number compared to how many shells exist. This thesis work will also look at more shells than the one from Greenberg et al. (2019), where different shells were looked at for POSIX-compliance (but more so as an afterthought). The ones already looked at by Greenberg will also be included, to see if anything has changed with updates to the respective shells since the paper was published.

3.1 Study aims

The first aim of this thesis work is to compare how different Linux shells do in terms of performance when it comes to scripting and common scripting functions.

The second aim of this thesis work is to compare how POSIX-compliant different Linux shells are. A lot of shells have POSIX-compliance as one of their goals, but as shown by Greenberg & Blatt (2019), the POSIX-conformance tests they used in the report came up with different amount of errors for different shells, meaning that there is some difference in actual POSIX-compliance, even if only in edge cases. Therefore, this thesis work aims to look at a lot of different modern shells, and see how they compare in the test, and also see in what areas of POSIX definition they fail.

The research questions are “How do different Linux shells compare in terms of performance?” and “How POSIX-compliant are different Linux shells?”

3.2 Motivation

This thesis work contributes to helping users of Linux and other UNIX-like operating systems who write scripts know the differences between different Linux shells, especially when it comes to scripting. Knowing which shells have better performance and in what areas, can be useful if a user works in an environment with shell scripts that run long or complicated functions that can take a lot of time.

Comparing the POSIX-compliance of different shells also helps with scripting portability, especially when it comes to some very specific and niche POSIX definitions that may not be implemented correctly in all POSIX-compliant shells. Seeing how POSIX-compliant each shell is can give an overview of how well script portability can be expected.

3.3 Delimitations

This work only looks at and compares command-line interface (CLI) shells, and not graphical user interface (GUI) shells. This is because GUI shells work very differently from CLI shells and are not even defined as “shells” in POSIX.

4 Methodology

Wohlin et al. (2012) describes different empirical methods to study the field of software engineering, which is relevant here as the aim of this report is to compare different software (shells) against each other. The methods listed are surveys, case studies, and experiments. The most fitting method for this work is a controlled experiment, as it allows us to study different "treatments" (shells in this case) in the same controlled setting to compare them against each other. Surveys or other methods which involve asking humans would not fit to answer the research question either, as it would only collect data on people's subjective experience on which shell is the fastest or most POSIX-compliant. Another method not listed but that could also be considered but did not fit is a systematic literature review. A systematic literature review would not work as there is not enough literature available to review on the topic of POSIX-compliance or performance.

The shells chosen for this work are: Bash, dash, yash, zsh, ksh, mksh, OSH, fish, tcsh. Bash, dash, yash, zsh, ksh, mksh (and OSH partially) were chosen as they were included in Greenberg & Blatt (2019). Looking at them again here makes it possible to both see how POSIX-compliance has changed since their study with these shells, and they can be studied in terms of performance, something Greenberg & Blatt did not. ksh, bash, zsh, tcsh, and fish were also chosen as they were used in Kidwai et al. (2021), where they compared these shells. They did not however compare these shells in terms of POSIX-compliance or performance, which is what the aim of this report is. Fish and tcsh were also chosen as these are shells that are popular (for being non-POSIX shells) but do not aim for POSIX compliance, this gives a point of comparison to see how non-POSIX-compliant shells compare to the ones that try to be.

4.1 Data collection

Data collection in this thesis work was done through running tests on different Linux shells in a virtual machine. This method was chosen as it allows for collecting data on up-to-date versions of each shell.

To collect data on POSIX-compliance, the tests created by Greenberg & Blatt (2019) to prove POSIX-compliance of Smoosh will be used, which are available on GitHub (Greenberg, 2023). The Smoosh test suite was created for Smoosh to test "obscure corners of the shell's behaviour". It can be run on any shell and is made up of different shell scripts that then check the output to see if it gives the excepted output or not. These tests will test some more obscure POSIX specification compared to the other test suites. The authors of the Smoosh paper also used the tests from the "Modernish" program, which is also available on GitHub (Dekker, 2024). Modernish is a library for writing POSIX compliant shell scripts, to help ensure that scripts are portable between different shells, Modernish has a test suite that it runs on the shell that the program is trying to be installed on. The tests report back the number of succeeded tests, as well as what bugs in the shell and "quirks" (unusual/uncommon behaviour unique to some shells, including behaviour that is against POSIX definition) that were identified. These tests can therefore be used as a way to see POSIX compatibility of different shells.

The POSIX tests will be run both on the latest available versions of the Smoosh and Modernish test suites, as well as the versions of the test suites that were used by Greenberg & Blatt (2019). This is because the amount of tests have changed since 2019, and running the tests the old version will make it possible to if/how much the different shells have improved in POSIX-compliance.

The program "Shellbench", available on GitHub (Nakashima, 2024), was used to compare performance between the different POSIX shells. It works by taking a file with a list of different defined benchmarks

which contains a command to continuously run as many times as possible for the specified benchmark time, the program then reports how many times per second each different shell was able to run the command. Different functions can therefore be tested on different shells to see which ones can run them most efficiently.

The fish and tcsh shells are not POSIX-compliant enough to work with Shellbench and return an error. To compare performance for these shells handmade scripts were made, which each run a different function that does the same thing on the shells a certain amount of times, and the time to run the scripts for each shell was then collected. All the Shellbench compatible shells were also tested with the handmade scripts to have a point of comparison.

4.2 Data analysis

To analyse the data, the results of the tests were collected for each of the shells and then compared against each other. For the POSIX tests (Smoosh test suite and Modernish test suite) the number of successful tests in each test suite was looked at, with a higher number of successful tests being better, i.e. more POSIX-compliant. Which tests failed in the Smoosh test suite and the bugs & quirks identified in the Modernish test suite is also presented. Which test(s) all shells struggle with was also analysed and presented. For the performance tests using Shellbench, the results were collected, presented in box plots, and then compared against each other, with a higher number meaning that the shell can run the function more times per second. For the handmade scripts for testing performance, the time to run each script for each shell was looked at with the "time" command, with a shorter time meaning that the shell can run the function faster. The times were then collected and presented in box plots.

4.3 Validity and Reliability

Wohlin et al. (2012) categorises validity threats when it comes to experimentation into four different categories: conclusion validity, internal validity, construct validity, and external validity. The validity threats in each category that are relevant to this work will be discussed, as well as how they were attempted to be mitigated.

4.3.1 Conclusion Validity

One threat to conclusion validity is *low statistical power*, meaning that there is not data to properly reveal a true pattern. To ensure that the statistical power is high enough the tests were run 30 times for the performance tests (the POSIX tests were not run more than twice as the result should not change as it is just testing features) to collect more data and to see if the results would differ from each run of the tests.

Another threat to conclusion validity is the *reliability of measures*, which deals with how reliable the measurements (i.e. the tests) are. Running the tests several times also helps mitigate this, as the results of the tests can be expected to be very similar or even the same each time it is run on each different shell.

The threat *random irrelevancies in experimental setting* is about factors outside the experimental setting affecting the result, such as a background program (e.g. package manager syncing) taking up a lot of processing power unexpectedly when one of the tests is being run. The host machine that the virtual machine for the tests were run on also contribute to this threat, as the machine was not purely used for virtualisation and could therefore be interfering with the performance of the virtual machine. To help mitigate this, the host machine was not used while running any of the performance tests.

4.3.2 Internal Validity

The internal threats listed by Wohlin et al. (2012) are not applicable to this work as they deal with experiments where humans are part of what is being tested on. This experiment only deals with computer shells which did not change and were not updated in the middle of the duration of the experiment (though it is possible that caching/temporary data stored by the shells could have some effect on the results).

4.3.3 Construct Validity

The threat *inadequate preoperational explication of constructs* deals with the constructs not being clearly defined enough beforehand so that it becomes unclear what the constructs, i.e. measurements, are actually measuring. This was mitigated by properly explaining the methodology and what is being looked at as in the problem formulation and data collection & analysis chapters above.

4.3.4 External Validity

One threat to external validity is the *interaction of setting and treatment*. This threat is about not having the setting (the shells and system) and the "treatment" (in this case measurement is being done instead of a treatment) be representative of actual what is actually used. This was mitigated by using Manjaro, which has very up-to-date packages of each shell, as well as each of the performance tests testing functions that are realistically used in scripting.

5 Implementation

This chapter describes the implementation of the tests and environment they were ran in.

5.1 Environment

The tests were run in a virtual machine running Manjaro Linux Zetar 25.0.0 using Oracle VirtualBox Version 7.1.6. The host machine has an AMD Ryzen 7 7800X3D processor with 16 cores running at 4.2GHz, and 64 GiB of RAM. The virtual machine was stored on an SSD and was allocated 16GiB of RAM, 6 cores of the processor, and a virtual disk of 50GiB.

The different shells were downloaded through the Manjaro repository using the “pacman” manager, or through the Arch User Repository (AUR) with the help of the AUR-helper “yay”. The shells that were downloaded through the Manjaro repository were: Bash, dash, OSH (package called oil), zsh, fish, and tcsh. The shells downloaded through the AUR were: mksh, ksh (package called ksh93-git. ksh is also available through the Manjaro repository, but that package is of an old abandoned version), and yash.

When Bash or yash is invoked as /bin/sh they will enter POSIX mode, where they try to conform to the POSIX definitions more strictly (Greenberg & Blatt, 2019). By default in Manjaro /bin/sh will be symbolically linked to Bash, to change this, the command “sudo ln -sf yash /bin/sh” can be run. The command can be run but with “Bash” instead of “yash” to but it back to default. This lets testing be performed on Bash and yash in their POSIX modes by specifying “sh” as the shell in the tests.

Table 2 show the shells that were used in the tests. The dates in the version number are part of the version name and not necessarily when the version was released:

Shell	Version
Bash	5.2.37(1)-release
dash	0.5.12-1
yash	2.58.1
fish	4.0.1
ksh	93u+m/1.1.0-alpha 2025-05-08
mksh	MIRBSD KSH R59 2020/10/31
zsh	5.9
tcsh	6.24.15 (Astron) 2025-02-04
OSH	Oils 0.26.0

Table 2: Version of Shells Used

5.2 POSIX Tests

The Smoosh test suite was downloaded by first making sure the dependencies were installed with pacman, listed in the table 3 along with the version that was installed:

Package	Version
gcc	14.2.1
make	4.4.1-2
autoconf	2.72-1
libtool	2.5.4
pkg-config	2.4.3-1
libffi	3.4.7-1

Table 3: Smoosh Dependencies

Once the dependencies are installed, the test suite was downloaded by cloning the Smoosh GitHub. Commit cc67dbe of Smoosh was used as the latest version, and commit 23e437f was used as the old version of the test suite, as it has the same amount of tests as in Greenberg & Blatt (2019) (the specific they used is not mentioned by Greenberg & Blatt in their report). After that, the tests can be accessed by opening a terminal in the Smoosh directory and running "TEST_SHELL=*shell* make -C tests", afterwards "make clean -C tests" can be ran to clear the cache. While running the test suite on some shells, the test suite would be temporarily suspended to the background. To continue the tests the command "fg" had to be entered into the command line, which brings the tests back to the foreground.

Modernish was downloaded through cloning the Modernish repository on GitHub. Commit e3c01f4 was used as the latest version, and commit ee3c548 was used as the old version as it has the same number of tests in the test suite as in the report by Greenberg & Blatt (2019). The tests can then be run by trying to install Modernish with the desired shell by running "./install.sh -s *shell*". The tests will then be run and after that the user will be asked if they want to use the shell as the default shell for Modernish, here CTRL+C can be pressed to cancel the actual installation and the tests can be run on another shell. To get Modernish to run all tests on the shell when attempting to install, the "install.sh" file needs to be edited on line 380: "if $msh_shell $MSH_PREFIX/bin/modernish --test -eqq; then". The "e" from "-eqq" here needs to be removed as this flag is set to "disable or reduce expensive (i.e. slow or memory-hogging) tests (Dekker, 2024, "Appendix B"). tcsh, fish, and OSH do not work with either version of Modernish and just return an error, therefore no test results from the Modernish test suite could be produced for these shells. Additionally, yash does not work with the 2019 version of the Modernish test suite and simply returns an error.

5.3 Performance tests

The following sections describe how Shellbench works, as well as how the handmade scripts were set up.

5.3.1 Shellbench

To set up the Shellbench tests, the Shellbench repository was cloned from GitHub (commit 4b04d18) and then no further installation was required. To run Shellbench the script was ran in the Shellbench directory with the command "./shellbench -w 10 -t 10 -s Bash,dash,sh,yash,ksh,mksh,zsh,OSH testscript.sh". The flags "-w 10" and "-t 10" specify the warm-up time and benchmark time respectively. The warm-up gives enough time between each test to make sure there are no residual effects from the previous test affecting the next test. The benchmark time specifies how long it should run each test for. The results are given in how many times per second the shell could run the benchmark, so increasing benchmark time increases the reliability of the result. The Shellbench application is actually a shell script starting with the shebang (which specifies what shell to run the script with) "#!/bin/sh", this works fine when Bash is linked to /bin/sh, but when testing yash in POSIX mode, yash is linked to /bin/sh, Shellbench stops working completely. To prevent this the shebang can be changed to "#!/bin/bash" when running the tests on yash in POSIX mode.

The test script provided to Shellbench can have several benchmarks in it. It is formatted as such:

```
#!/bin/sh

#bench "name"
@begin
command
@end
```

The first line defines the name of the benchmark, then the command(s) that should be ran is put between "@begin" and "@end".

The test script that was used to test every shell includes five benchmarks:

```
#!/bin/sh

#bench “noop”
@begin
:
@end

#bench “increment variable”
@begin
i=$((i+1))
@end

#bench “subshell”
@begin
(
          true
)
@end

#bench “command substitution”
@begin
var=$(true)
@end

#bench “external command”
@begin
/usr/bin/true
@end
```

The “noop” benchmark runs the “no operation” function, which does nothing, this essentially just tests how many times the shell can interpret a command per second. The “increment variable” increments a integer variable by one, testing the shells integer manipulation speed. The “subshell” benchmark runs the “true” command (which does nothing, successfully) inside a subshell to test how efficiently each shell can create and kill subshells. The “command substitution” benchmark tests command substitution by setting the output of the “true” command as the value of the “var” variable. Using the “true” command for subshell and command substitution helps test the actual functions as true doesn’t do anything and therefore only the subshell operation is what is being measured. Finally, the “external command” benchmark runs the external “true” command (not the built-in version which every shell has) which tests how efficiently each shell can access external applications.

5.3.2 Handmade Scripts

The handmade scripts for test similar functions. They work by looping a command 100000 (100000 is used and not a higher number because fish returns an error when a higher number is tried), the script is then run with the “time” command to see how long each shell takes to run it, this is then repeated 30 times. The scripts with the “bash” shebang also work with all the other POSIX-compatible shells by

simply changing the shebang (e.g. #!/bin/dash/ instead of #!/bin/bash).

The first set of scripts runs “no operation” (noop) with the “:” command.

```
#!/bin/bash
for i in {1..100000}; do
        :
done
```

```
#!/bin/fish
for i in (seq 1 100000)
        :
end
```

```
#/bin/tcsh
foreach i (`seq 1 100000`)
        :
end
```

The second set of scripts increment an integer variable by 1 until it reaches 100001.

```
#!/bin/bash
a=1
for i in $(seq 1 100000); do
        a=$((a+1))
done
```

```
#!/bin/fish
set a 1
for i in (seq 1 100000)
        set a (math ”$a +1”)
end
```

```
#/bin/tcsh
set a=1
foreach i (`seq 1 100000`)
        @ a++
end
```

The final set of scripts tests command substitution by setting the value of the “var” variable to the output of the “true” command.

```
#!/bin/Bash
for i in $(seq 1 100000); do
          var=$(true)
done
```

```
#!/bin/fish
for i in (seq 1 100000)
          set var (true)
end
```

```
#!/bin/tcsh
foreach i (`seq 1 100000`)
          set var=`true`
end
```

6 Results

In this chapter the results of the tests are presented.

6.1 POSIX Tests

The results from the Smoosh test suite are presented in the table 4. The number of successful tests are out of a total of 186 tests. The test suite was run at least twice to ensure that the result was the same for each run with a specific shell.

Shell	Bash	Bash (sh)	dash	yash	yash (sh)	fish	ksh	mksh	zsh	tcsh	OSH
Successful Tests	152	158	150	151	152	27	145	154	132	31	140

Table 4: Results from 2023 Smoosh Test Suite

The tests that fail on all the shells are presented below. The tests can be found in the Smoosh GitHub repository in the "tests" folder, with the files ending in .test being the test that the shell runs, and .out, .err, and .ec files being the STDOUT, STDERR, and error code expected from the shells (Greenberg, 2023). A list of all the tests failed on each shell can be found in appendix A.

```
shell/builtin.break.nonlexical
shell/builtin.command.nospecial
shell/builtin.continue.nonlexical
shell/builtin.dot.nonexistent
shell/builtin.history.nonposix
shell/builtin.source.nonexistent
shell/builtin.times.ioerror
shell/builtin.trap.subshell.false.exit
shell/builtin.trap.subshell.loud2
shell/builtin.trap.subshell.loud
shell/builtin.trap.subshell.ture.ec1
shell/builtin.unset
shell/semantics.error.noninteractive
shell/semantics.return.trap
```

The results from the old Smoosh test suite from 2019 are presented in the table 5. The number of successful tests are out of a total of 161.

Shell	Bash	Bash (sh)	dash	yash	yash (sh)	fish	ksh	mksh	zsh	tcsh	OSH
Successful Tests	129	135	125	127	128	24	122	131	112	29	118

Table 5: Results from 2019 Smoosh Test Suite

The results from the latest Modernish test suite are presented in table 6. The number of successful tests are out of a total of 385 tests. Even when removing the "-e" option from the install script, Modernish still skips some tests for unknown reasons, this is shown in the "skipped tests" row. The amount of failed tests as reported by the test suite is shown in the "Failed Tests" row. The "quirks" and "bugs" rows show the amount of quirks and bugs identified by the test suite when running. A list of all the quirks and bugs identified for each shell can be found in appendix B. The test suite was also run twice here on each shell to ensure the result was the same on each shell.

Shell	Bash	Bash (sh)	dash	yash	yash (sh)	ksh	mksh	zsh
Successful Tests	382	382	372	377	372	379	377	374
Skipped Tests	3	3	9	8	13	6	6	3
Failed Tests	0	0	3	0	0	0	2	8
Quirks	2	3	3	10	9	5	2	6
Bugs	0	0	2	0	0	0	2	3

Table 6: Results from 2024 Modernish Test Suite

Results from the old version of the Modernish test suite is presented in table 7. The number of successful tests are out of a total 312 tests. Note that yash did not work with the old version of Modernish and produced an error.

Shell	Bash	Bash (sh)	dash	yash	yash (sh)	ksh	mksh	zsh
Successful Tests	310	310	300	X	X	305	305	308
Skipped Tests	2	2	9	X	X	4	5	4
Failed Tests	0	0	3	X	X	3	2	2
Quirks	4	4	2	X	X	3	2	4
Bugs	0	0	2	X	X	0	1	1

Table 7: Results from 2019 Modernish Test Suite

6.2 Performance tests

The results from Shellbench tests are presented in the following box plots. Each test was run thirty times. The values on the Y axis represent how many times per second the shell was able to run the benchmark (higher is better).

The results from the “no operation” tests in figure 1:

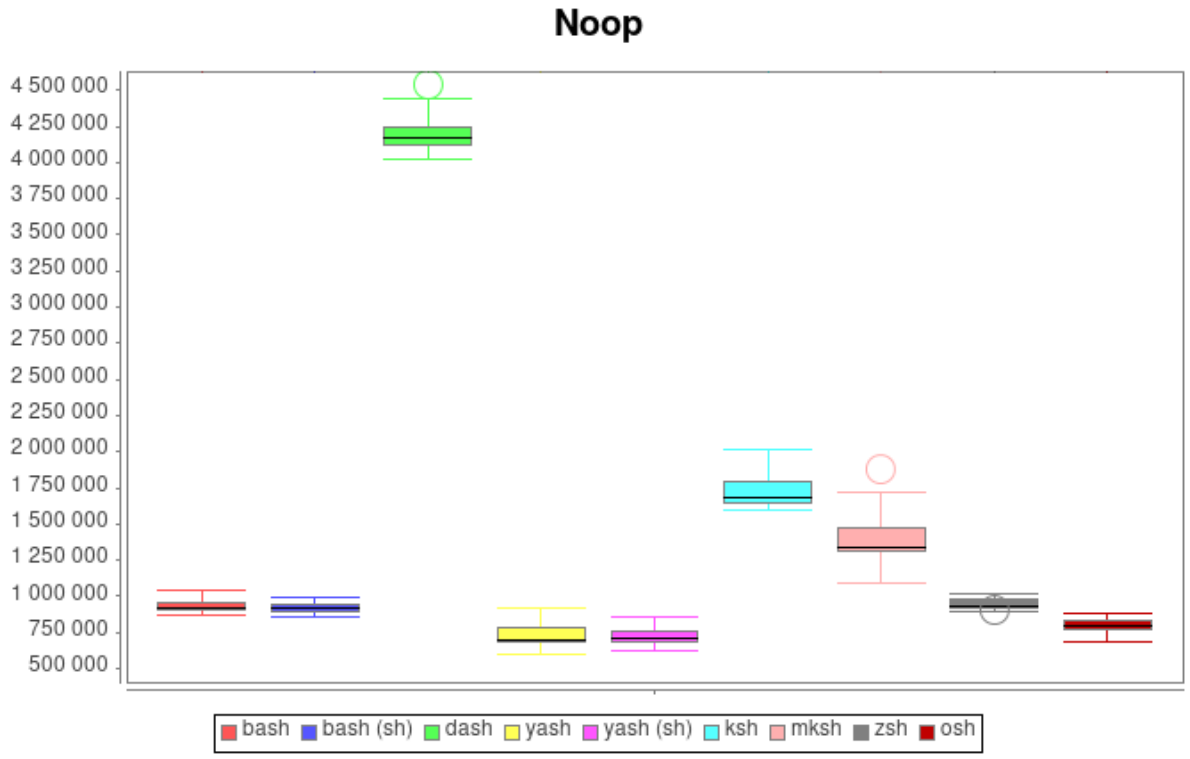

Figure 1: Shellbench Noop Tests

The results from the “increment variable” tests in figure 2:

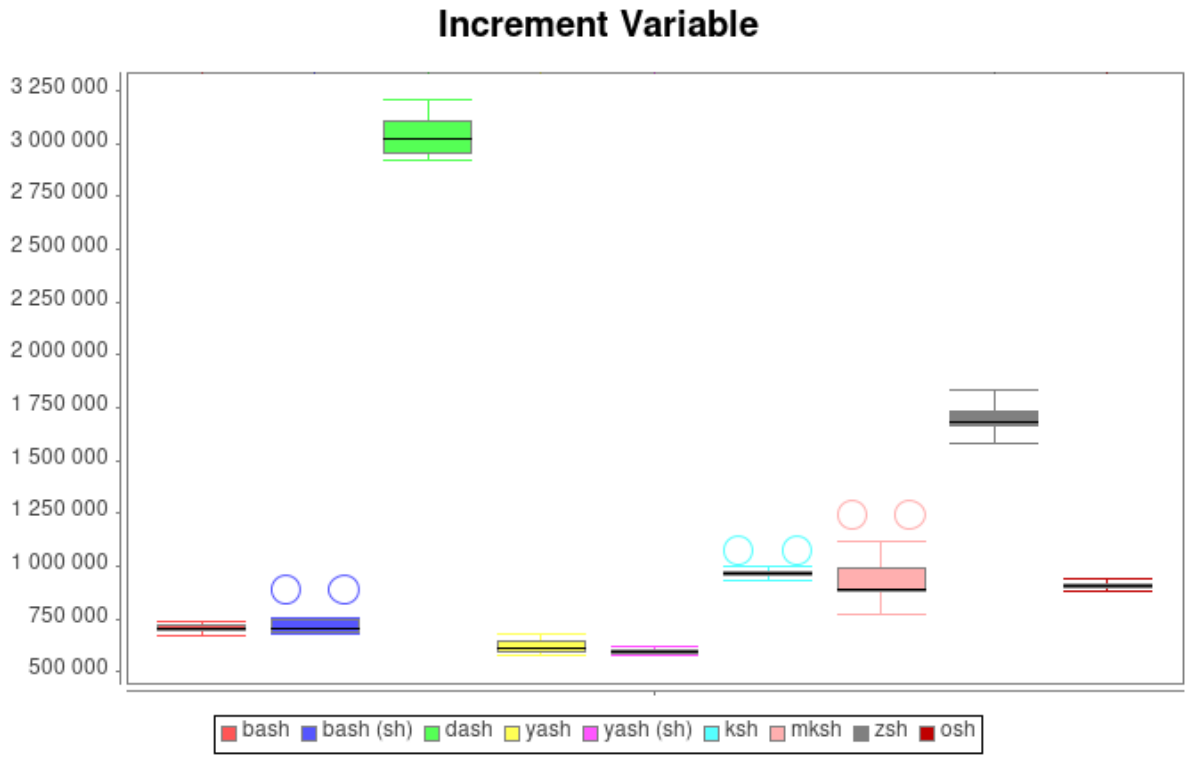

Figure 2: Shellbench Increment Variable Tests

The results from the “Subshell” tests. Two box plots are given, one with (figure 3) and one without ksh (figure 4), since ksh is very far away from the other shells, the box plot gets too zoomed out to see the difference between the other shells:

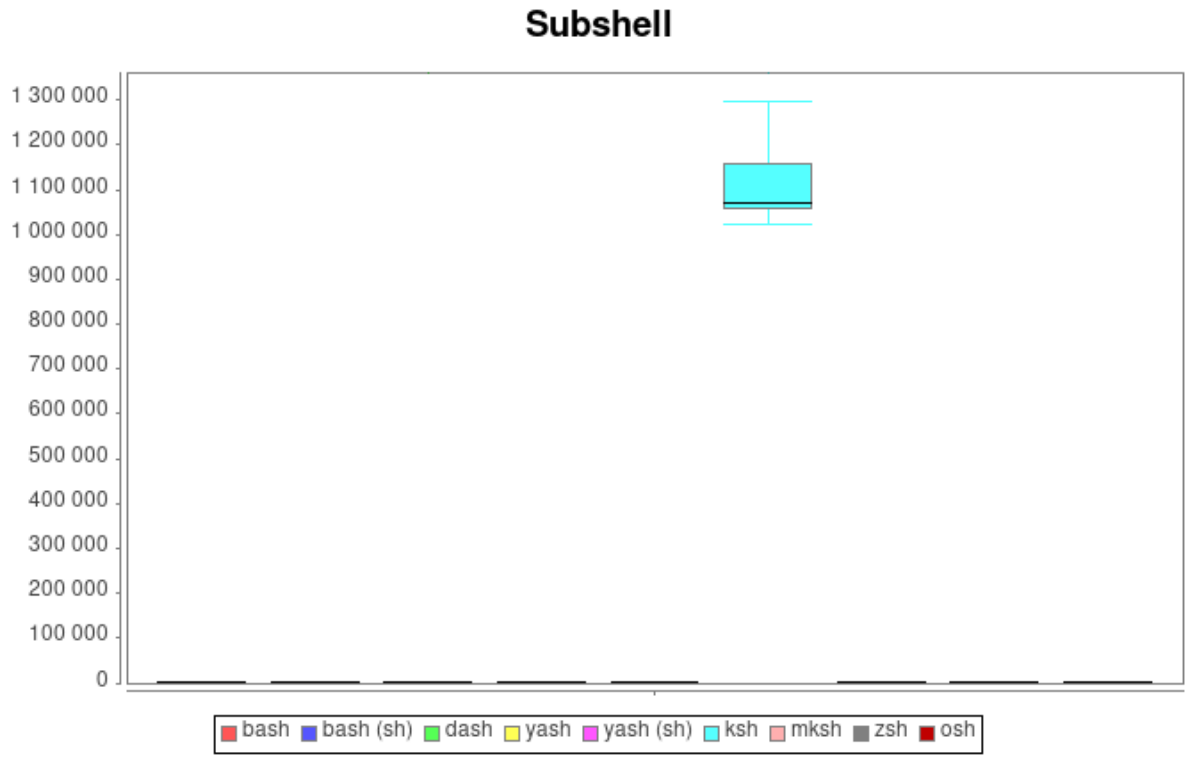

Figure 3: Shellbench Subshell Tests

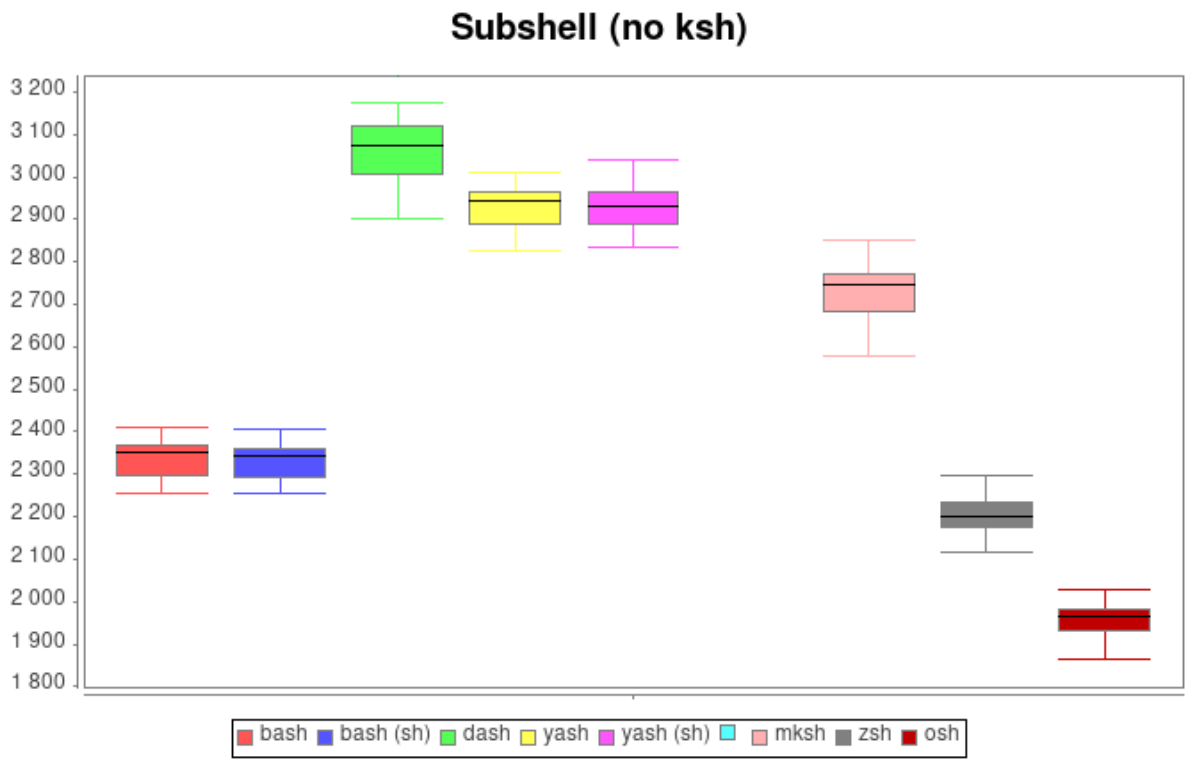

Figure 4: Shellbench Subshell Tests (no ksh)

The results from the “command substitution” tests, two box plots are given here again for the same reason as above, one with (figure 5) and one without ksh (figure 6):

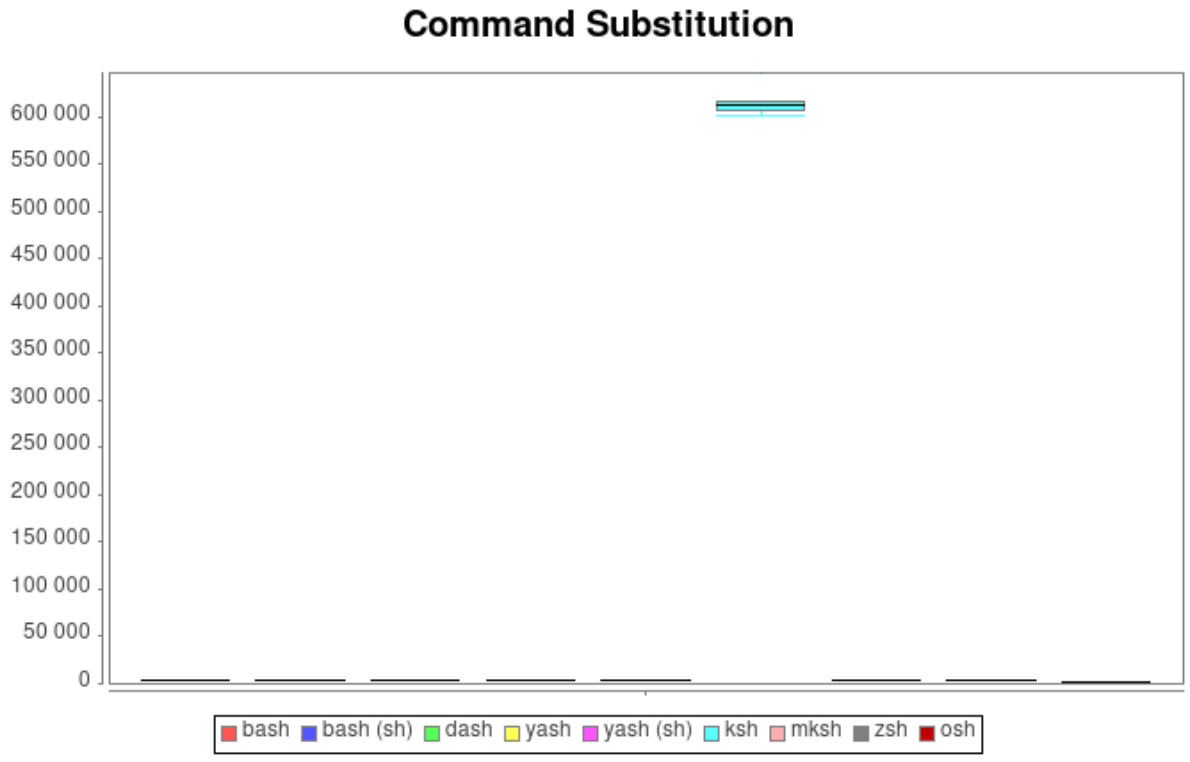

Figure 5: Shellbench Command Substitution Tests

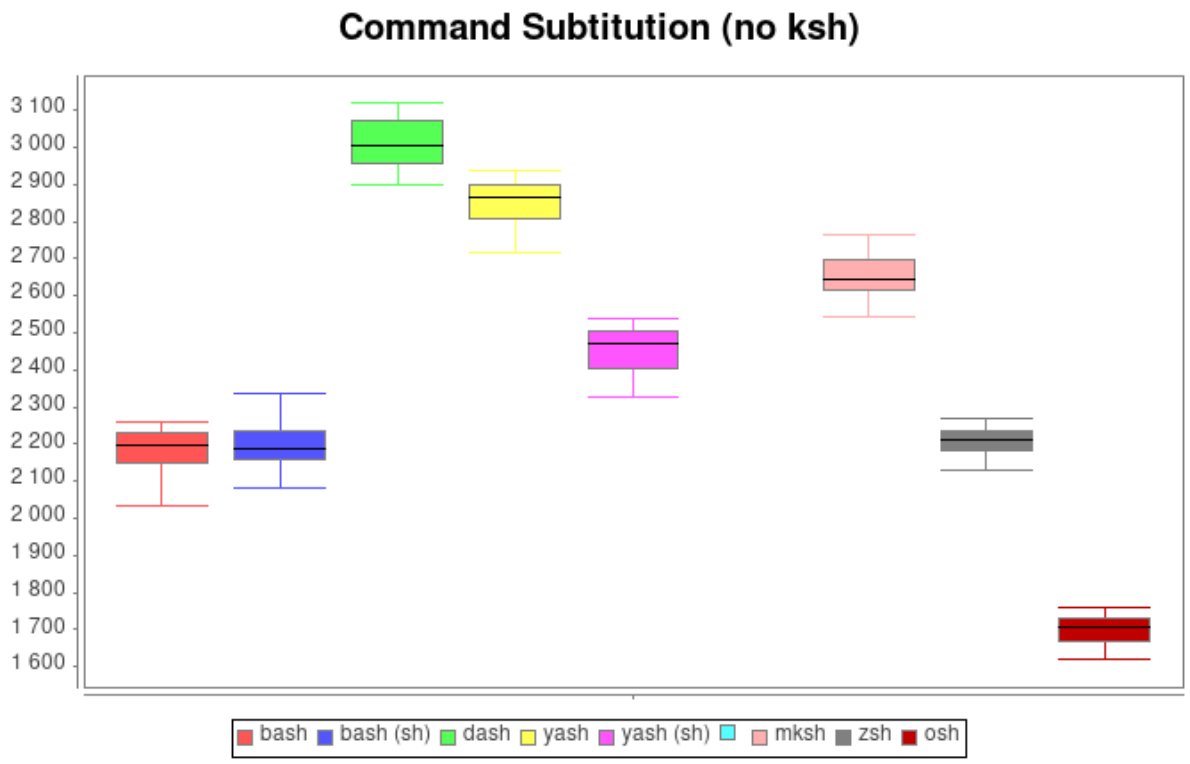

Figure 6: Shellbench Command Substitution Tests (no ksh)

The results from “external command” tests in figure 7:

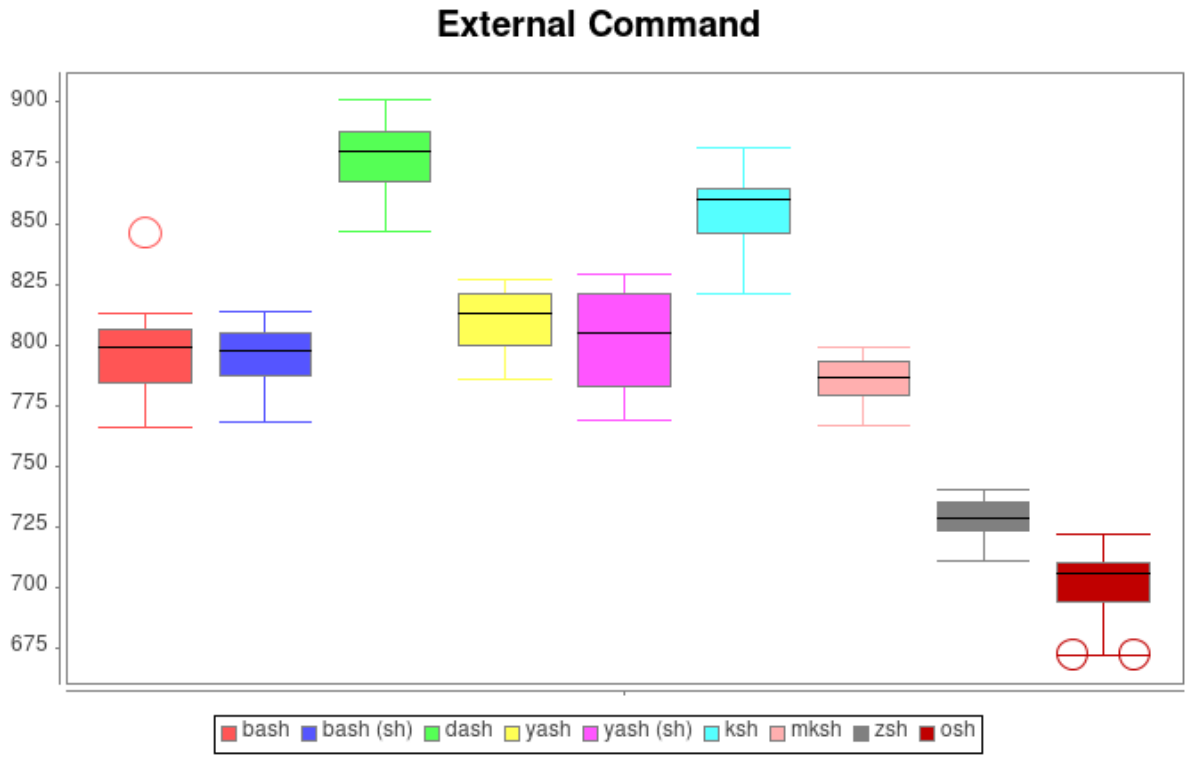

Figure 7: Shellbench External Command Tests

The results from the handmade scripts are presented in the following box plots (figure 8, 9, and 10). The values on the Y axis represent the amount of seconds taken to run each script as reported by the time command in Manjaro (lower is better). As there was not much variation between each run, the boxes are all very thin.

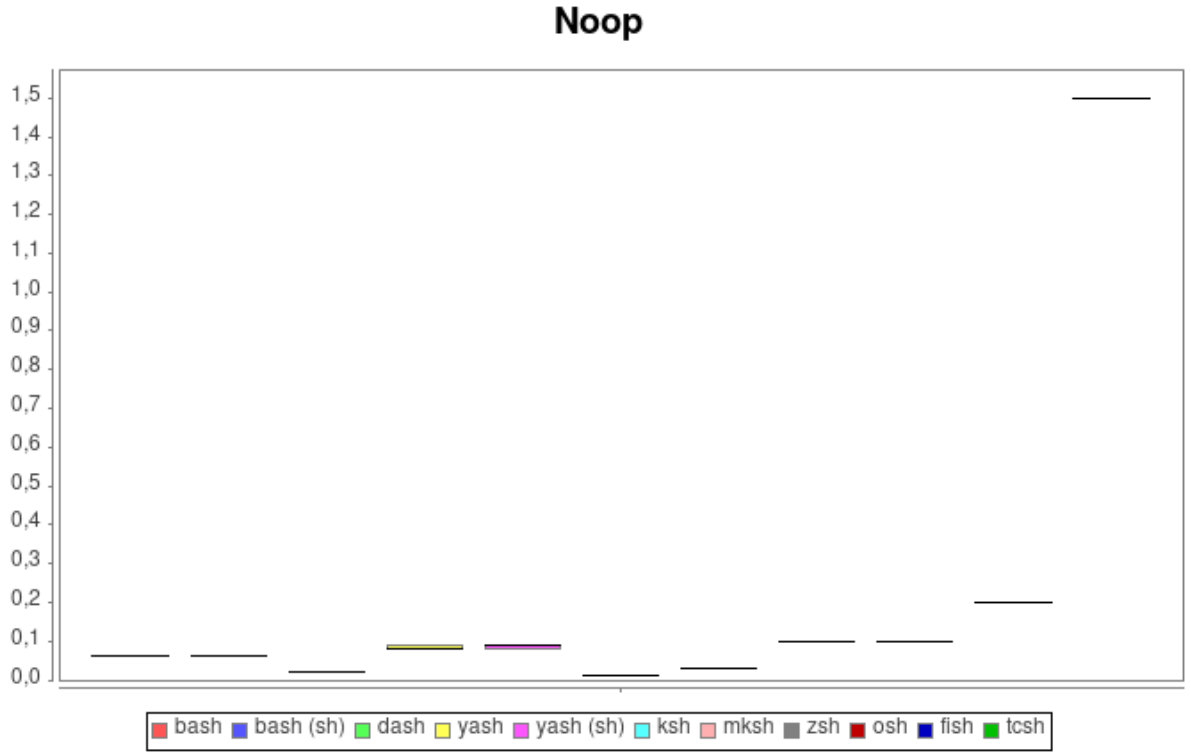

Figure 8: Handmade Noop Tests

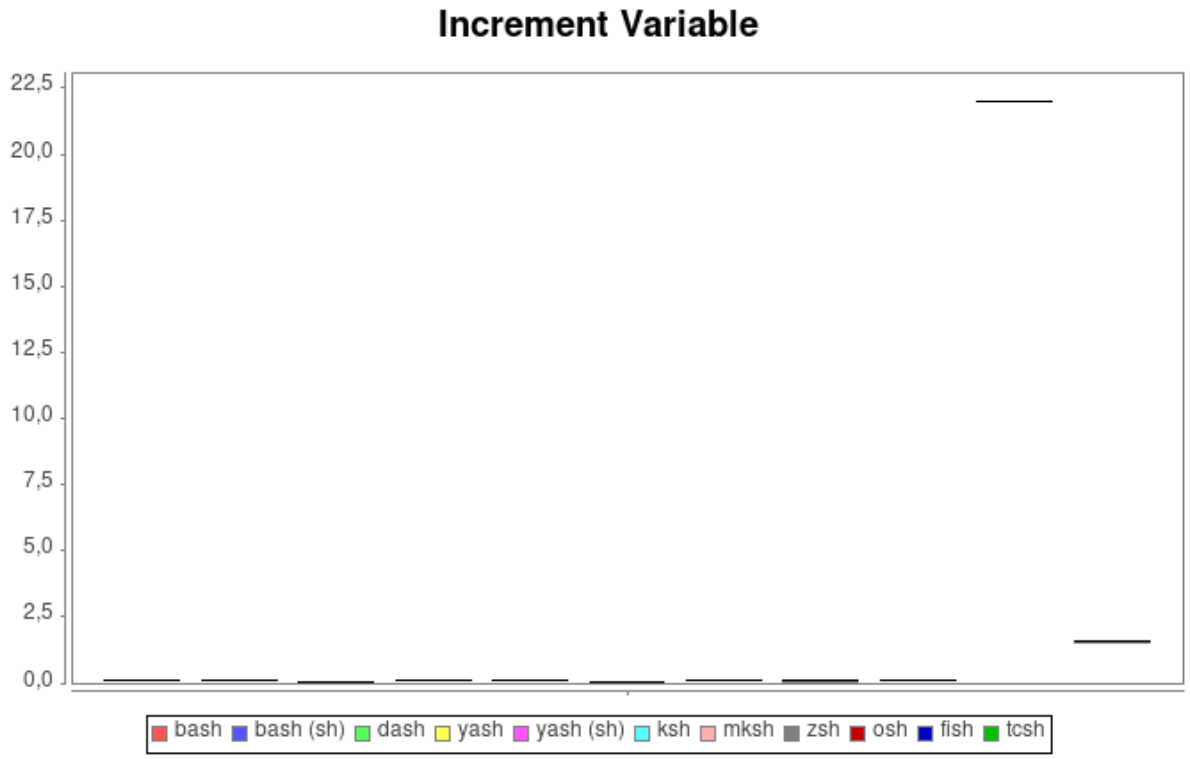

Figure 9: Handmade Increment Variable Tests

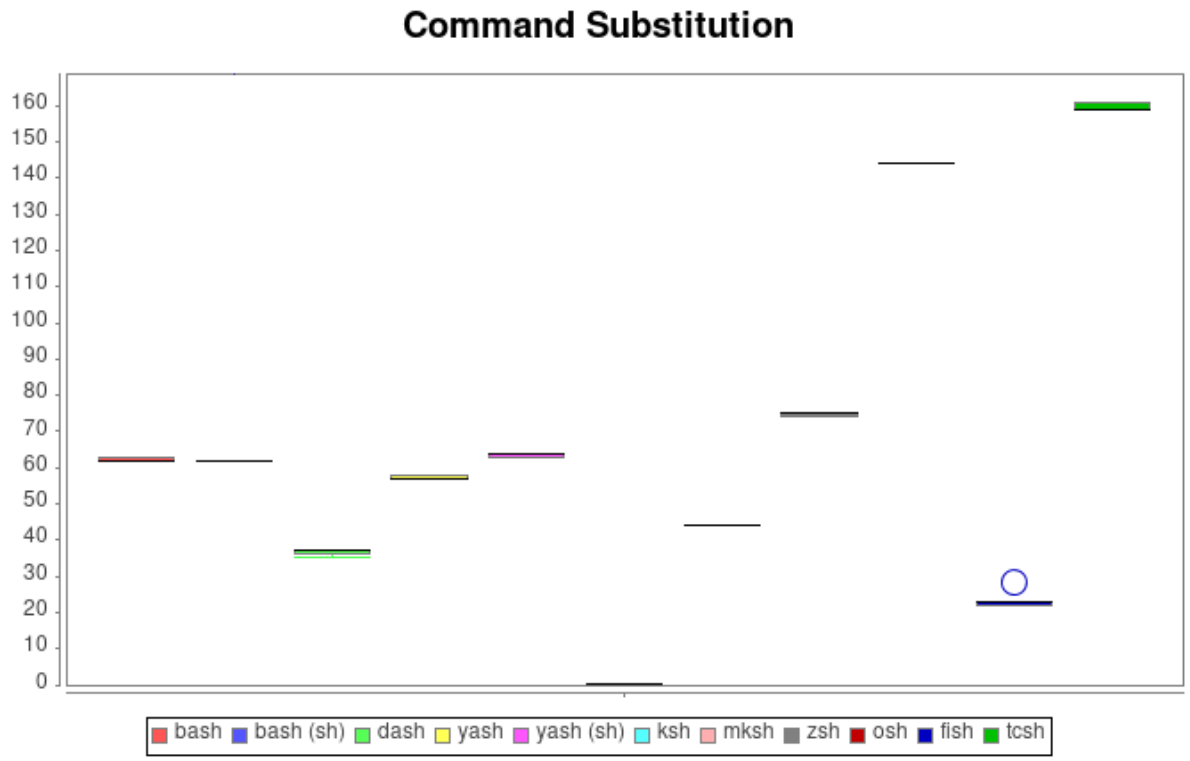

Figure 10: Handmade Command Substitution Tests

7 Discussion

From the results of the POSIX tests the shell that passed the most tests in both the Smoosh and Modernish test suite was Bash in POSIX mode (158/186 in the Smoosh test suite, and 0 failing tests with 3 quirks in Modernish). The shells that do *not* aim to be POSIX-compliant, fish and tcsh, did very badly in the Smoosh test suite (fish 27/186 and tcsh 31/186), and was unable to even run the Modernish test suite which is telling of its POSIX incompatibility. Zsh performed the worst in the Smoosh test suite (132/186), not counting fish and tcsh. Generally, the spread between how well shells performed on the POSIX-compliance tests was not that big (not counting fish and tcsh) and they were all between 132-158 successful tests out of the 186 total, which points to these shells having good compatibility. As Modernish skips some of the tests even with the removal of the "-e" flag, the results from Modernish become less reliable. No information on why these tests were skipped could be found, and it is possible that if these tests did run that some shells would be performing worse/better than they are right now in the tests.

In terms of the subshell and command substitution performance tests, ksh was able to run the test command more than 200 times more per second than any other shell. This is due to the fact ksh tries to not create an actual subprocess where possible to save on performance, but the output will still be processes as if it ran inside a subshell (Robbins & Rosenblatt, 2002). It is therefore highly likely that ksh is not creating actual new processes when the subprocess and command substitution tests (which creates a subshell) are running and is therefore able to run the command way faster.

From the Shellbench results one shell stand out as always being at the top, dash. Dash is always first except for in the tests involving subshells (due to the non-forking of ksh), where it comes in second. For the "noop" and "increment variable" tests dash is faster than the other shells by quite a large margin (the average is at least a million higher than second place).

From the results of the handmade scripts which included the shells incompatible with Shellbench, the results are quite mixed. Dash was the fastest in the "noop" and "increment variable" tests, ksh once again wins out on command substitution followed by fish. This is due to fish working similar to ksh when it comes to command substitution, in that it does not create a separate process when creating subshells (Fish Shell, n.d-a). Tcsh performs very badly in the command substitution and noop tests, and only does not perform the worst in the "increment variable" test because fish performed badly. The results point at the Shellbench-incompatible shells being slower performance wise when compared to the Shellbench-compatible shells.

7.1 In relation to previous research

Table 8 below shows the results of the Smoosh test suite from the different versions as well as the results that Greenberg & Blatt (2019) got when they ran the tests. This allows for comparison between old and new versions of the shells, to see how much more POSIX-compliant the shells have gotten with updates.

Shell	Bash	Bash (sh)	dash	yash	yash (sh)	ksh	mksh	zsh	OSH
Successful Tests (2023 version) out of a total 186	152	158	150	151	152	145	154	145	140
Successful Tests (2019 version) out of a total 161	129	135	125	127	128	122	131	112	118
Successful Tests (Greenberg & Blatt) out of a total 161	126	131	119	122	118	120	127	109	105

Table 8: Number of successful tests from different versions of Smoosh test suite

When POSIX-compliance is compared to what Greenberg & Blatt (2019) found six years ago in their study, compliance has generally increased. Greenberg & Blatt ran the Smoosh and Modernish test suite on Bash, dash, zsh, mksh, ksh, and yash. For example, when they ran their tests Bash succeeded 131 tests in POSIX mode, whereas now it passes 135. OSH has increased the most since 2019 in amount of passing tests, going from 105 to 118, an increase of 13. Overall, all the shells tested in 2019 have increased in number of passing tests, pointing to shells moving toward greater POSIX-compliance with updates.

Table 9 below shows results of the Modernish test suite from the different versions and compared to the report from Greenberg & Blatt (2019). The results are shown in “tests failed” as this is what is shown in Greenberg & Blatt’s report, without showing how many tests were skipped.

Shell	Bash	Bash (sh)	dash	yash	yash (sh)	ksh	mksh	zsh
Failed Tests (2024 version) out of a total 385	0	0	3	0	0	0	2	8
Failed Tests (2019 version) out of a total 312	0	0	3	X	X	3	2	3
Failed Tests (Greenberg & Blatt) out of a total 312	20	20	3	1	1	17	3	3

Table 9: Results from different versions of Modernish test suite

The trend of improvement is also clear when comparing the results of the Modernish test suite; Greenberg & Blatt found Bash to fail 20 tests in Modernish, whereas now it only fails zero. Dash however has seen no improvement since 2019, and still fails three tests, even in the latest 2024 version of Modernish. Modernish skipping tests makes the results more unreliable however, especially as Greenberg & Blatt (2019) do not mention tests skipping at all and do not go in depth with how they ran the Modernish tests. Also important to remember that Modernish does not strictly test how POSIX-compliant shells are, but rather how Modernish-compatible they are.

Kidwai et al. (2021) looked at different shells and compared them each other. Their findings were that zsh was the most outstanding shell in terms of what they compared the shells against each other in, which was interactive, safety, and scripting features. When comparing that to the results of the POSIX and performance tests, zsh failed the most amount of tests in both Modernish and the Smoosh test suite out of the POSIX-compliant shells, and during the performance tests it did not stand out in any of the tests. This might point to zsh being good for interactive and personal use, but less so for performance and scripting portability.

7.2 Ethical and societal aspects

The results of this study could be used by actors with bad actors, such as hackers, to know which shell would be most exploitable if scripts need to be executed fast (for example before a user has time to shut down the computer if they notice the attack), or to know if malicious scripts from one shell would be compatible with another, which could assist in assessing if a system is vulnerable to attack due to the system using a compatible shell.

8 Conclusion

The research questions for this work were “How do different Linux shells compare in terms of performance?” and “How POSIX-compliant are different Linux shells?” To answer this, this work has looked at 9 different Linux shells and compared them in terms of POSIX-compliance and performance. The results of the study found that Linux shells that aim to be POSIX-compliant generally are except in some niche cases, and the ones that are not (fish and tcsh), are not compliant in most cases. When comparing old test results to new ones a clear trend of improved POSIX-compliance can also be seen in newer version of shells. When it comes to performance dash stands out as the fastest shell, except in cases where a lot of subshells are called, where ksh and fish beats all the others due to their unique optimisation by avoiding forking to create new processes when subshells would usually be created. In general the POSIX-compliant shells are faster than the non-POSIX-compliant shells.

The results of this study can be used by users, and especially script programmers, of Linux and other UNIX-like operating systems to easier pick what shell to use. The results have shown that dash is outstanding in cases were performance is important, except for cases where subshells are used frequently, in which case ksh might be a preferable pick. When it comes to scripting compatibility between shells that aim for POSIX compatibility, the different shells tested seem to have a pretty equal amount of POSIX-compliance and compatibility can therefore be expected (if writing scripts with only POSIX defined functions), but may fail in some edge-cases.

8.1 Future work

For future work one could focus more on just one of the research questions of this study, for example only looking at POSIX-compliance of shells and seeing what functions specifically are unsupported, or only looking at performance and measuring more functions as well as having a bigger sample size. More shells could also be included in future work such as more shells that do not aim for POSIX-compliance, e.g. elvish, nushell, PowerShell (which is available on Linux but is mainly for Windows), xonsh, and ion. Comparing these shells might be hard due to their inherent differences.

To further expand on POSIX testing, the official POSIX test suite from The Open Group could be used. This test suite is meant to test operating systems, shells and utilities for POSIX-compliance for official POSIX certification. The Open Group was contacted during the writing of this thesis asking for permission to use it, however no response was given.

References

Apple. (2023). *Use Zsh as the default shell on your Mac*. Apple Support. https://support.apple.com/en-us/102360

Comparison of computer shells. (2025, May 13). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Comparison_of_command_shells&oldid=1290187861

Computerwoche (2009, March 4). *The A-Z of Programming Languages: Bourne shell* Computerwoche.com. Retrieved March 20, 2025, from https://www.computerwoche.de/article/2720775/the-a-z-of-programming-languages-bourne-shell.html

Debian Project. (2025). *10. Files*. Debian Policy Manual (Version 4.7.2.0). Retrieved 2025, March 31 from https://www.debian.org/doc/debian-policy/ch-files.html

Dekker, M. (2024). M*odernish*. GitHub. https://github.com/modernish/modernish

Fish Shell. (n.d.-a). *Design.* Fish-shell 4.0.1 documentation. Retrieved 2025, March 27, from https://fishshell.com/docs/current/design.html

Fish Shell. (n.d.-b). *Fish for Bash users*. Fish-shell 4.0.1 documentation. Retrieved 2025, March 27, from https://fishshell.com/docs/current/fish_for_Bash_users.html

Free Software Foundation. (2022). *Bash(1) manual page (GNU Bash 5.2). Retrieved 2025*. Retrieved 2025, March 31 from https://www.man7.org/linux/man-pages//man1/Bash.1.html

Greenberg, M. (2023). *Smoosh.* https://github.com/mgree/smoosh

Greenberg, M., & Blatt, A. J. (2019). Executable formal semantics for the POSIX shell. *Proceedings of the ACM on Programming Languages*, *4*(POPL), 1–30. https://doi.org/10.1145/3371111

Greenberg, M., Kallas, K., & Vasilakis, N. (2021). Unix shell programming. *Proceedings of the Workshop on Hot Topics in Operating Systems*, 104–111. https://doi.org/10.1145/3458336.3465294

Isaak, J. (1990). Standards—the history of POSIX: A study in the standards process. *Computer*, 23(7), 89-92. https://doi.org/10.1109/2.56856

Jones, M. (2011, December 8). *Evolution of shells in Linux*. IBM developer. https://developer.ibm.com/tutorials/l-linux-shells/

Kidwai, A., Arya, C., Singh, P., Diwakar, M., Singh, S., Sharma, K., & Kumar, N. (2021) A comparative study on shells in Linux: A review. *Materials Today: Proceedings*, Volume 37, Part 2, 2021, pp. 2612-2616, https://doi.org/10.1016/j.matpr.2020.08.508.

Machado, A. (2024, November 9). *Evolution of unix shells: A comprehensive guide to SH, bash, fish, zsh, CSH, and Ksh*. Evolution of Unix Shells: A Comprehensive Guide to sh, bash, fish, zsh, csh, and ksh. https://machaddr.substack.com/p/evolution-of-unix-shells-a-comprehensive

Maschek, S. (2021). *Ash (Almquist Shell) Variants*. Retrieved 2025, March 28, from https://www.in-ulm.de/~mascheck/various/ash/

MirBSD. (n.d.). *Mksh-the mirbsd korn shell*. http://mirbsd.de/mksh

Nakashima, K. (2024). *ShellBench*. GitHub. https://github.com/shellspec/shellbench

Pickle, B. (2023, March 8). *Shell*. TechTerms.com. Retrieved March 9, 2025, from

https://techterms.com/definition/shell

Robbins, A. & Rosenblatt, B. (2002). *Learning the Korn Shell* (2nd ed.). O'Reilly Media, Inc.

The IEEE and The Open Group. (2024). "POSIX.1-2024: The Open Group Base Specifications Issue 8". *IEEE.* https://pubs.opengroup.org/onlinepubs/9799919799/

Ubuntu Wiki. (2017). *DashAsBinSh*. Ubuntu Wiki. Retrieved 2025, March 28, from https://wiki.ubuntu.com/DashAsBinSh

Watanabe, Y. (2025). *Yash.* GitHub. https://github.com/magicant/yash

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). *Experimentation in Software Engineering*. Springer. https://doi.org/10.1007/978-3-642-29044-2

Oilshell. (2023). *Oils 2023 FAQ*. https://www.oilshell.org/blog/2023/03/faq.html

Z Shell. (2010). *Shell frequently-asked questions*. Retrieved 2025, June 20, from https://zsh.sourceforge.io/FAQ/

Appendices

Appendix A – Failed Smoosh Tests

The following is a list of all the tests in the Smoosh (latest 2023 version) test suite each shell failed. The tests can be found in the Smoosh GitHub repository in the "tests" folder, with the files ending in .test being the test that the shell runs, and .out and .err files being the expected STDOUT and STDERR from the shells (Greenberg, 2023).

Bash:
shell_tests.sh: 152/186 tests passed
shell_tests.sh failing tests: shell/builtin.alias.empty shell/builtin.break.nonlexical shell/builtin.command.nospecial shell/builtin.continue.nonlexical shell/builtin.dot.break shell/builtin.dot.nonexistent shell/builtin.exitcode shell/builtin.export.unset shell/builtin.history.nonposix shell/builtin.kill.jobs shell/builtin.readonly.assign.noninteractive shell/builtin.source.nonexistent.earlyexit shell/builtin.source.nonexistent shell/builtin.special.redir.error shell/builtin.times.ioerror shell/builtin.trap.return shell/builtin.trap.subshell.false.exit shell/builtin.trap.subshell.loud2 shell/builtin.trap.subshell.loud shell/builtin.trap.subshell.true.ec1 shell/builtin.unset shell/parse.eval.error shell/semantics.dot.glob shell/semantics.error.noninteractive shell/semantics.evalorder.fun shell/semantics.-h.nonposix shell/semantics.interactive.expansion.exit shell/semantics.return.trap shell/semantics.special.assign.visible.nonposix shell/semantics.subshell.background.traps shell/semantics.tilde.sep shell/semantics.traps.inherit shell/sh.interactive.ps1 shell/sh.ps1.override

Bash (sh):
shell_tests.sh: 158/186 tests passed
shell_tests.sh failing tests: shell/builtin.break.nonlexical shell/builtin.command.nospecial shell/builtin.continue.nonlexical shell/builtin.dot.break shell/builtin.dot.nonexistent shell/builtin.exitcode shell/builtin.history.nonposix shell/builtin.kill.jobs shell/builtin.source.nonexistent shell/builtin.times.ioerror shell/builtin.trap.exitcode shell/builtin.trap.return shell/builtin.trap.subshell.false.exit shell/builtin.trap.subshell.loud2 shell/builtin.trap.subshell.loud shell/builtin.trap.subshell.true.ec1 shell/builtin.unset shell/semantics.dot.glob shell/semantics.error.noninteractive shell/semantics.evalorder.fun shell/semantics.-h.nonposix shell/semantics.interactive.expansion.exit shell/semantics.return.trap shell/semantics.subshell.background.traps shell/semantics.tilde.sep shell/semantics.traps.inherit shell/sh.interactive.ps1 shell/sh.ps1.override

dash:
shell_tests.sh: 150/186 tests passed
shell_tests.sh failing tests: shell/builtin.break.nonlexical shell/builtin.command.nospecial shell/builtin.continue.nonlexical shell/builtin.dot.break shell/builtin.dot.nonexistent shell/builtin.dot.path shell/builtin.exec.badredir shell/builtin.history.nonposix shell/builtin.jobs shell/builtin.readonly.assign.interactive shell/builtin.readonly.assign.noninteractive shell/builtin.source.nonexistent.earlyexit shell/builtin.source.nonexistent shell/builtin.source.setvar shell/builtin.special.redir.error shell/builtin.test.-nt.-ot.absent shell/builtin.times.ioerror shell/builtin.trap.chained shell/builtin.trap.exitcode shell/builtin.trap.subshell.false.exit

shell/builtin.trap.subshell.loud2 shell/builtin.trap.subshell.loud shell/builtin.trap.subshell.true.ec1
shell/builtin.trap.supershell shell/builtin.unset shell/semantics.error.noninteractive shell/semantics.-
h.nonposix shell/semantics.interactive.expansion.exit shell/semantics.noninteractive.expansion.exit
shell/semantics.redir.close shell/semantics.return.trap shell/semantics.subshell.background.traps
shell/semantics.subshell.break shell/semantics.traps.inherit shell/semantics.var.dashu
shell/sh.ps1.override

Yash:
shell_tests.sh: 151/186 tests passed
shell_tests.sh failing tests: shell/builtin.break.nonlexical shell/builtin.command.nospecial
shell/builtin.continue.nonlexical shell/builtin.dot.break shell/builtin.dot.nonexistent
shell/builtin.exec.badredir shell/builtin.history.nonposix shell/builtin.readonly.assign.noninteractive
shell/builtin.source.nonexistent.earlyexit shell/builtin.source.nonexistent shell/builtin.source.setvar
shell/builtin.special.redir.error shell/builtin.test.numeric.spaces.nonposix shell/builtin.times.ioerror
shell/builtin.trap.chained shell/builtin.trap.kill.undef shell/builtin.trap.return
shell/builtin.trap.subshell.false.exit shell/builtin.trap.subshell.loud2 shell/builtin.trap.subshell.loud
shell/builtin.trap.subshell.true.ec1 shell/builtin.unset shell/parse.error shell/semantics.arith.pos
shell/semantics.case.escape.quotes shell/semantics.error.noninteractive
shell/semantics.interactive.expansion.exit shell/semantics.noninteractive.expansion.exit
shell/semantics.redir.close shell/semantics.return.trap shell/semantics.simple.link
shell/semantics.traps.inherit shell/semantics.wait.alreadydead shell/sh.interactive.ps1
shell/sh.ps1.override

Yash (sh):
shell_tests.sh: 152/186 tests passed
shell_tests.sh failing tests: shell/builtin.break.lexical shell/builtin.break.nonlexical
shell/builtin.command.nospecial shell/builtin.continue.lexical shell/builtin.continue.nonlexical
shell/builtin.dot.break shell/builtin.dot.nonexistent shell/builtin.exec.badredir
shell/builtin.history.nonposix shell/builtin.source.nonexistent.earlyexit shell/builtin.source.nonexistent
shell/builtin.source.setvar shell/builtin.special.redir.error shell/builtin.test.numeric.spaces.nonposix
shell/builtin.times.ioerror shell/builtin.trap.chained shell/builtin.trap.exitcode
shell/builtin.trap.kill.undef shell/builtin.trap.return shell/builtin.trap.subshell.false.exit
shell/builtin.trap.subshell.loud2 shell/builtin.trap.subshell.loud shell/builtin.trap.subshell.true.ec1
shell/builtin.unset shell/parse.error shell/semantics.case.escape.quotes
shell/semantics.error.noninteractive shell/semantics.noninteractive.expansion.exit
shell/semantics.redir.close shell/semantics.return.trap shell/semantics.simple.link
shell/semantics.traps.inherit shell/semantics.wait.alreadydead shell/sh.ps1.override

fish:
shell_tests.sh: 27/186 tests passed
shell_tests.sh failing tests: shell/benchmark.fact5 shell/benchmark.while shell/builtin.alias.empty
shell/builtin.break.lexical shell/builtin.break.nonlexical shell/builtin.cd.pwd shell/builtin.command.ec
shell/builtin.command.exec shell/builtin.command.keyword shell/builtin.command.nospecial
shell/builtin.command.special.assign shell/builtin.continue.lexical shell/builtin.continue.nonlexical
shell/builtin.dot.break shell/builtin.dot.nonexistent shell/builtin.dot.path shell/builtin.dot.return
shell/builtin.eval.break shell/builtin.eval shell/builtin.exec.badredir

shell/builtin.exec.modernish.mkfifo.loop shell/builtin.exitcode shell/builtin.export.override
shell/builtin.export shell/builtin.export.unset shell/builtin.history.nonposix shell/builtin.jobs
shell/builtin.kill0_+5 shell/builtin.kill0 shell/builtin.kill.jobs shell/builtin.kill.signame
shell/builtin.readonly.assign.interactive shell/builtin.readonly.assign.noninteractive
shell/builtin.set.quoted shell/builtin.source.nonexistent.earlyexit shell/builtin.source.nonexistent
shell/builtin.source.setvar shell/builtin.special.redir.error shell/builtin.times.ioerror
shell/builtin.trap.chained shell/builtin.trap.exit3 shell/builtin.trap.exitcode
shell/builtin.trap.exit.subshell shell/builtin.trap.false shell/builtin.trap.nested shell/builtin.trap.redirect
shell/builtin.trap.return shell/builtin.trap.subshell.false shell/builtin.trap.subshell.loud2
shell/builtin.trap.subshell.loud shell/builtin.trap.subshell.quiet shell/builtin.trap.subshell.true.ec1
shell/builtin.trap.subshell.truefalse shell/builtin.trap.supershell shell/builtin.unset shell/parse.emptyvar
shell/parse.error shell/parse.eval.error shell/semantics.arith.assign.multi
shell/semantics.arithmetic.bool_to_num shell/semantics.arithmetic.tilde
shell/semantics.arith.modernish shell/semantics.arith.pos shell/semantics.arith.var.space
shell/semantics.assign.noglob shell/semantics.assign.visible shell/semantics.background.nojobs.stdin
shell/semantics.background.pid shell/semantics.background.pipe.pid shell/semantics.background
shell/semantics.backtick.exit shell/semantics.backtick.fds shell/semantics.backtick.ppid
shell/semantics.case.ec shell/semantics.case.escape.modernish shell/semantics.case.escape.quotes
shell/semantics.command.argv0 shell/semantics.command-subst.newline shell/semantics.command-
subst shell/semantics.-C shell/semantics.defun.ec shell/semantics.dot.glob
shell/semantics.errexit.carryover shell/semantics.errexit.subshell shell/semantics.errexit.trap
shell/semantics.error.noninteractive shell/semantics.escaping.backslash.modernish
shell/semantics.escaping.backslash shell/semantics.escaping.heredoc.dollar
shell/semantics.escaping.newline shell/semantics.escaping.quote shell/semantics.escaping.single
shell/semantics.eval.makeadder shell/semantics.evalorder.fun
shell/semantics.expansion.heredoc.backslash shell/semantics.expansion.quotes.adjacent
shell/semantics.expansion.substring shell/semantics.for.readonly shell/semantics.fun.error.restore
shell/semantics.-h.nonposix shell/semantics.ifs.combine.ws shell/semantics.interactive.expansion.exit
shell/semantics.kill.traps shell/semantics.length shell/semantics.monitoring.ttou shell/semantics.no-
command-subst shell/semantics.noninteractive.expansion.exit shell/semantics.pattern.bracket.quoted
shell/semantics.pattern.hyphen shell/semantics.pattern.modernish shell/semantics.pattern.rightbracket
shell/semantics.pipe.chained shell/semantics.redir.close shell/semantics.redir.fds
shell/semantics.redir.indirect shell/semantics.redir.toomany shell/semantics.return.and
shell/semantics.return.if shell/semantics.return.not shell/semantics.return.or shell/semantics.return.trap
shell/semantics.return.while shell/semantics.simple.link shell/semantics.slash.glob
shell/semantics.special.assign.visible.nonposix shell/semantics.splitting.ifs
shell/semantics.subshell.background.traps shell/semantics.subshell.break
shell/semantics.subshell.redirect shell/semantics.subshell.return2 shell/semantics.subshell.return
shell/semantics.substring.quotes shell/semantics.tilde.colon shell/semantics.tilde.no-exp
shell/semantics.tilde.quoted.prefix shell/semantics.tilde.quoted shell/semantics.tilde.sep
shell/semantics.tilde shell/semantics.traps.async shell/semantics.traps.inherit
shell/semantics.var.alt.nullifs shell/semantics.var.alt.null shell/semantics.varassign
shell/semantics.var.builtin.nonspecial shell/semantics.var.format.tilde
shell/semantics.variable.escape.length shell/semantics.var.ifs.sep shell/semantics.var.star.emptyifs
shell/semantics.var.star.format shell/semantics.var.unset.nofield shell/semantics.wait.alreadydead
shell/semantics.while shell/sh.-c.arg0 shell/sh.env.ppid shell/sh.interactive.ps1 shell/sh.monitor.bg

shell/sh.monitor.fg shell/sh.ps1.override shell/sh.set.ifs

ksh:
shell_tests.sh: 145/186 tests passed
shell_tests.sh failing tests: shell/builtin.break.nonlexical shell/builtin.command.nospecial
shell/builtin.continue.nonlexical shell/builtin.dot.break shell/builtin.dot.nonexistent
shell/builtin.exitcode shell/builtin.history.nonposix shell/builtin.jobs shell/builtin.kill.jobs
shell/builtin.readonly.assign.interactive shell/builtin.source.nonexistent.earlyexit
shell/builtin.source.nonexistent shell/builtin.times.ioerror shell/builtin.trap.chained
shell/builtin.trap.exit3 shell/builtin.trap.subshell.false.exit shell/builtin.trap.subshell.loud2
shell/builtin.trap.subshell.loud shell/builtin.trap.subshell.true.ec1 shell/builtin.unset
shell/parse.eval.error shell/semantics.background.nojobs.stdin shell/semantics.background
shell/semantics.dot.glob shell/semantics.errexit.trap shell/semantics.error.noninteractive
shell/semantics.evalorder.fun shell/semantics.-h.nonposix shell/semantics.interactive.expansion.exit
shell/semantics.kill.traps shell/semantics.redir.fds shell/semantics.return.trap
shell/semantics.subshell.background.traps shell/semantics.subshell.break
shell/semantics.tilde.quoted.prefix shell/semantics.tilde.sep shell/semantics.traps.inherit
shell/semantics.wait.alreadydead shell/sh.monitor.bg shell/sh.monitor.fg shell/sh.ps1.override

mksh:
shell_tests.sh: 154/186 tests passed
shell_tests.sh failing tests: shell/builtin.break.nonlexical shell/builtin.command.nospecial
shell/builtin.continue.nonlexical shell/builtin.dot.nonexistent shell/builtin.exitcode
shell/builtin.history.nonposix shell/builtin.kill.jobs shell/builtin.readonly.assign.interactive
shell/builtin.readonly.assign.noninteractive shell/builtin.source.nonexistent.earlyexit
shell/builtin.source.nonexistent shell/builtin.times.ioerror shell/builtin.trap.chained
shell/builtin.trap.exitcode shell/builtin.trap.return shell/builtin.trap.subshell.false.exit
shell/builtin.trap.subshell.loud2 shell/builtin.trap.subshell.loud shell/builtin.trap.subshell.true.ec1
shell/builtin.trap.supershell shell/builtin.unset shell/semantics.dot.glob
shell/semantics.error.noninteractive shell/semantics.-h.nonposix
shell/semantics.interactive.expansion.exit shell/semantics.kill.traps shell/semantics.redir.fds
shell/semantics.return.trap shell/semantics.splitting.ifs shell/semantics.subshell.break
shell/semantics.var.dashu shell/semantics.wait.alreadydead

zsh:
shell_tests.sh: 132/186 tests passed
shell_tests.sh failing tests: shell/builtin.break.lexical shell/builtin.break.nonlexical
shell/builtin.command.exec shell/builtin.command.nospecial shell/builtin.continue.lexical
shell/builtin.continue.nonlexical shell/builtin.dot.break shell/builtin.dot.nonexistent
shell/builtin.dot.path shell/builtin.exec.modernish.mkfifo.loop shell/builtin.exitcode
shell/builtin.export.unset shell/builtin.history.nonposix shell/builtin.kill.jobs shell/builtin.set.quoted
shell/builtin.source.nonexistent.earlyexit shell/builtin.source.nonexistent
shell/builtin.special.redir.error shell/builtin.test.-nt.-ot.absent
shell/builtin.test.numeric.spaces.nonposix shell/builtin.times.ioerror shell/builtin.trap.nested
shell/builtin.trap.subshell.false.exit shell/builtin.trap.subshell.loud2 shell/builtin.trap.subshell.loud
shell/builtin.trap.subshell.true.ec1 shell/builtin.trap.subshell.truefalse shell/builtin.trap.supershell

shell/builtin.unset shell/parse.emptyvar shell/parse.error shell/parse.eval.error
shell/semantics.arith.var.space shell/semantics.backtick.fds shell/semantics.dot.glob
shell/semantics.error.noninteractive shell/semantics.escaping.backslash.modernish
shell/semantics.escaping.quote shell/semantics.-h.nonposix shell/semantics.interactive.expansion.exit
shell/semantics.pattern.hyphen shell/semantics.pattern.modernish shell/semantics.pattern.rightbracket
shell/semantics.quote.backslash shell/semantics.return.if shell/semantics.return.trap
shell/semantics.return.while shell/semantics.special.assign.visible.nonposix
shell/semantics.splitting.ifs shell/semantics.subshell.break shell/semantics.tilde.quoted.prefix
shell/sh.interactive.ps1 shell/sh.ps1.override shell/sh.set.ifs

tcsh:
shell_tests.sh: 31/186 tests passed
shell_tests.sh failing tests: shell/benchmark.fact5 shell/benchmark.while shell/builtin.alias.empty
shell/builtin.break.lexical shell/builtin.break.nonlexical shell/builtin.cd.pwd shell/builtin.command.ec
shell/builtin.command.exec shell/builtin.command.keyword shell/builtin.command.nospecial
shell/builtin.command.special.assign shell/builtin.continue.lexical shell/builtin.continue.nonlexical
shell/builtin.dot.break shell/builtin.dot.nonexistent shell/builtin.dot.path shell/builtin.dot.return
shell/builtin.dot.unreadable shell/builtin.echo.exitcode shell/builtin.eval.break shell/builtin.eval
shell/builtin.exec.modernish.mkfifo.loop shell/builtin.exitcode shell/builtin.export.override
shell/builtin.export shell/builtin.export.unset shell/builtin.hash.nonposix shell/builtin.history.nonposix
shell/builtin.jobs shell/builtin.kill0_+5 shell/builtin.kill.jobs shell/builtin.kill.signame
shell/builtin.pwd.exitcode shell/builtin.readonly.assign.interactive
shell/builtin.readonly.assign.noninteractive shell/builtin.set.-m shell/builtin.set.quoted
shell/builtin.source.nonexistent shell/builtin.source.setvar shell/builtin.special.redir.error
shell/builtin.times.ioerror shell/builtin.trap.chained shell/builtin.trap.exit3 shell/builtin.trap.exitcode
shell/builtin.trap.exit.subshell shell/builtin.trap.false shell/builtin.trap.kill.undef
shell/builtin.trap.nested shell/builtin.trap.redirect shell/builtin.trap.return
shell/builtin.trap.subshell.false shell/builtin.trap.subshell.loud2 shell/builtin.trap.subshell.loud
shell/builtin.trap.subshell.true.ec1 shell/builtin.trap.subshell.truefalse shell/builtin.trap.supershell
shell/builtin.unset shell/parse.emptyvar shell/parse.error shell/semantics.arith.assign.multi
shell/semantics.arithmetic.bool_to_num shell/semantics.arithmetic.tilde
shell/semantics.arith.modernish shell/semantics.arith.pos shell/semantics.arith.var.space
shell/semantics.assign.noglob shell/semantics.assign.visible shell/semantics.background.nojobs.stdin
shell/semantics.background.pid shell/semantics.background.pipe.pid shell/semantics.background
shell/semantics.backtick.exit shell/semantics.backtick.fds shell/semantics.backtick.ppid
shell/semantics.case.ec shell/semantics.case.escape.modernish shell/semantics.case.escape.quotes
shell/semantics.command.argv0 shell/semantics.command-subst.newline shell/semantics.-C
shell/semantics.defun.ec shell/semantics.dot.glob shell/semantics.errexit.carryover
shell/semantics.errexit.subshell shell/semantics.errexit.trap shell/semantics.error.noninteractive
shell/semantics.escaping.backslash.modernish shell/semantics.escaping.heredoc.dollar
shell/semantics.escaping.quote shell/semantics.escaping.single shell/semantics.eval.makeadder
shell/semantics.evalorder.fun shell/semantics.expansion.substring shell/semantics.for.readonly
shell/semantics.fun.error.restore shell/semantics.-h.nonposix shell/semantics.ifs.combine.ws
shell/semantics.interactive.expansion.exit shell/semantics.kill.traps shell/semantics.length
shell/semantics.no-command-subst shell/semantics.pattern.bracket.quoted
shell/semantics.pattern.hyphen shell/semantics.pattern.modernish shell/semantics.pattern.rightbracket

shell/semantics.pipe.chained shell/semantics.redir.fds shell/semantics.redir.from
shell/semantics.redir.indirect shell/semantics.redir.nonregular shell/semantics.redir.toomany
shell/semantics.redir.to shell/semantics.return.and shell/semantics.return.if shell/semantics.return.not
shell/semantics.return.or shell/semantics.return.trap shell/semantics.return.while
shell/semantics.simple.link shell/semantics.slash.glob shell/semantics.special.assign.visible.nonposix
shell/semantics.splitting.ifs shell/semantics.subshell.background.traps shell/semantics.subshell.break
shell/semantics.subshell.redirect shell/semantics.subshell.return2 shell/semantics.subshell.return
shell/semantics.substring.quotes shell/semantics.tilde.colon shell/semantics.tilde.no-exp
shell/semantics.tilde.quoted.prefix shell/semantics.tilde.quoted shell/semantics.tilde.sep
shell/semantics.tilde shell/semantics.traps.async shell/semantics.traps.inherit
shell/semantics.var.alt.nullifs shell/semantics.var.alt.null shell/semantics.varassign
shell/semantics.var.builtin.nonspecial shell/semantics.var.format.tilde
shell/semantics.variable.escape.length shell/semantics.var.ifs.sep shell/semantics.var.star.emptyifs
shell/semantics.var.star.format shell/semantics.var.unset.nofield shell/semantics.wait.alreadydead
shell/semantics.while shell/sh.-c.arg0 shell/sh.env.ppid shell/sh.interactive.ps1 shell/sh.monitor.bg
shell/sh.monitor.fg shell/sh.ps1.override shell/sh.set.ifs

OSH:
shell_tests.sh: 140/186 tests passed
shell_tests.sh failing tests: shell/builtin.alias.empty shell/builtin.break.lexical
shell/builtin.break.nonlexical shell/builtin.command.nospecial shell/builtin.continue.lexical
shell/builtin.continue.nonlexical shell/builtin.dot.break shell/builtin.dot.nonexistent
shell/builtin.dot.path shell/builtin.export.unset shell/builtin.history.nonposix shell/builtin.jobs
shell/builtin.kill.jobs shell/builtin.printf.repeat shell/builtin.readonly.assign.interactive
shell/builtin.source.nonexistent.earlyexit shell/builtin.source.nonexistent
shell/builtin.special.redir.error shell/builtin.times.ioerror shell/builtin.trap.kill.undef
shell/builtin.trap.nested shell/builtin.trap.redirect shell/builtin.trap.subshell.false.exit
shell/builtin.trap.subshell.loud2 shell/builtin.trap.subshell.loud shell/builtin.trap.subshell.true.ec1
shell/builtin.trap.supershell shell/builtin.unset shell/parse.error shell/parse.eval.error
shell/semantics.backtick.exit shell/semantics.error.noninteractive shell/semantics.-h.nonposix
shell/semantics.interactive.expansion.exit shell/semantics.kill.traps shell/semantics.redir.nonregular
shell/semantics.return.trap shell/semantics.special.assign.visible.nonposix
shell/semantics.subshell.break shell/semantics.traps.async shell/semantics.var.star.emptyifs
shell/semantics.wait.alreadydead shell/sh.interactive.ps1 shell/sh.monitor.bg shell/sh.monitor.fg
shell/sh.ps1.override

Appendix B – Modernish Quirks and Bugs

The following is a list of quirks and bugs identified by the Modernish (latest version) test suite for each shell. Explanation for what each quirk and bug means can be found on the Modernish GitHub page (Dekker, 2024).

Bugs for Bash:
QRK_EMPTPPWRD
QRK_UNSETF

Bugs for Bash (sh):
QRK_EMPTPPWRD
QRK_SPCBIXP
QRK_UNSETF

Bugs for dash:
BUG_LNNONEG
BUG_LOOPRET2
QRK_EVALNOOPT
QRK_LOCALINH
QRK_GLOBDOTS

Bugs for yash:
QRK_ARITHEMPT
QRK_OPTCASE
QRK_EMPTPPWRD
QRK_OPTDASH
QRK_GLOBDOTS
QRK_OPTNOPRFX
QRK_LOCALUNS
QRK_OPTULINE
QRK_OPTABBR
QRK_SPCBIXP

Bugs for yash (sh):
QRK_EMPTPPWRD
QRK_OPTCASE
QRK_OPTNOPRFX
QRK_SPCBIXP
QRK_GLOBDOTS
QRK_OPTDASH
QRK_OPTULINE
QRK_ARITHWHSP
QRK_OPTABBR

Bugs for ksh:
QRK_EMPTPPWRD
QRK_OPTABBR
QRK_OPTDASH
QRK_OPTNOPRFX

QRK_OPTULINE

Bugs for mksh:
BUG_MULTIBIFS
BUG_HDOCMASK
QRK_LOCALUNS
QRK_32BIT

Bugs for zsh:
BUG_ZSHNAMES
BUG_ARITHSPLT
BUG_TRAPUNSRE
QRK_BCDANGER