Monch: A Structured Shell for Linux

William GOODALL < owo@gatech.edu >
Flynt WALLACE < uwu@gatech.edu >

October 8, 2021

1 Abstract

Monch (Meta Object Notation Compact sHell) is a structure-oriented shell for Linux, and a set of compatible replacements for several standard Unix coreutils. A traditional shell generally relies on text: programs communicate with streams of text, environment variables contain text, and variables (for the most part) are stored as text. Because of this, most shell programs generally involve a lot of string parsing. Instead, monch deals in structured data—a superset of JSON, including lists, maps, and scalar values. This will help avoid complex, slow, and error-prone string operations, which are hard to write interactively, and hard to maintain in a script.

Contents

2 Motivation

The Unix shell is one of the oldest programs every command-line user will use every single day. The history of Unix shells is long and complicated, but through the years, most shells have not changed that much: they invoke a program, connect standard in to your keyboard, connect standard out to your terminal, and directly connect multiple processes together with pipes.

Monch will work differently. In general, the input, output, and arguments to a command will be structured. The shell will use MessagePack—an extensible, binary encoding of JSON—to pass data into and out of its child processes. Monch will ship with modified versions of the standard Unix coreutils and shell builtins (like `ls`, `grep`, and `cut`) that all understand and output MessagePack-encoded data. Finally, monch will provide some syntax that makes this data easy to store and manipulate (like extracting a single field in the middle of a pipeline, for instance).

This will not change any Linux process semantics—most commands will be normal Linux executables, found on the `$PATH`, which happen to speak MessagePack on `stdin`, on `stdout`, and in their arguments. Because of this, we don't require any modifications to the host operating system.

The main challenge this shell will solve is compatibility—what, exactly, to do with programs that don't output structured data in the way monch expects. Monch will have to support a set of useful coercions between structured and unstructured formats, so that other shell programs remain useful. Monch will have to convert a newline-delimited text stream into a stream of MessagePack strings on standard in, for instance, and vice versa for an incompatible program's output. Monch will also include several explicit conversion operators—like a JSON pretty-printer, CSV parser, and so on.

These tools should all come together to make a shell that's a lot more useful for complex interactive and scripted tasks.

3 Literature Review

This concept is not new by any means. A few shells do this already, or at least include concepts that are very similar to what we're planning to implement in monch.

3.1 PowerShell

Perhaps the most obvious prior art in this space is Microsoft's PowerShell.[3] PowerShell is based on the .NET Common Language Runtime, which describes a

shared object system usable from C#, C++, and other languages. PowerShell's commands (called "cmdlets") operate on these objects—exposing the entire .NET standard library to scripting.[4]

This is a powerful model—but works much better on Windows than it does Unix-based systems. Before PowerShell, Windows (and DOS before it) had a fairly bad shell experience through `cmd.exe`. Because of this, most Windows systems leaned heavily on system APIs, instead of the famous Linux model where "everything is a file." PowerShell extends this, bringing those same system API calls to the shell, and focusing less on file I/O. One example of this is PowerShell's ability to "cd" into the Windows registry—presenting a different set of Windows API calls instead of file operations.[5]

It's because of this that the PowerShell model is a bad fit for Linux. When most operations can be carried out by editing a text file, Unix shells have excelled—because they live and breathe by their ability to shunt streams of text around between processes.

Monch is an attempt to bridge this gap, as a Linux shell which aims to expose more powerful tools which are still manipulating streams of data at the end of the day.

3.2 Oil Shell

Oil is an alternative shell for Linux which aims to fix some of the syntax issues with a traditional POSIX-compatible shell. Mainly, Oil resolves some of the problems with POSIX `sh`'s quoting-and-splitting mechanisms.[6]

For instance, in POSIX `sh`, the following script is valid:

```
# Get a space-separated list of all directories
# starting with 'dir_glob_'
$ dir_to_remove="$(ls dir_glob_*)"

# Delete that directory
$ rm -rf $dir_to_remove/
```

However, it contains a really nasty bug—if no directories exist matching the glob, the `$dir_to_remove` variable will contain the empty string. The `rm` command will expand literally to `rm -rf /`. This command will delete every file on your system![1]

Oil resolves that by using a slightly-stricter grammar for shell commands—one that doesn't implicitly string-split command arguments, or expand variables implicitly inside strings.[1]

Monch will take inspiration from Oil, by parsing command lines according to a grammar, and performing any substitutions and splits only when the user explicitly asks for it. We feel that this is a safer design, less prone to footguns.

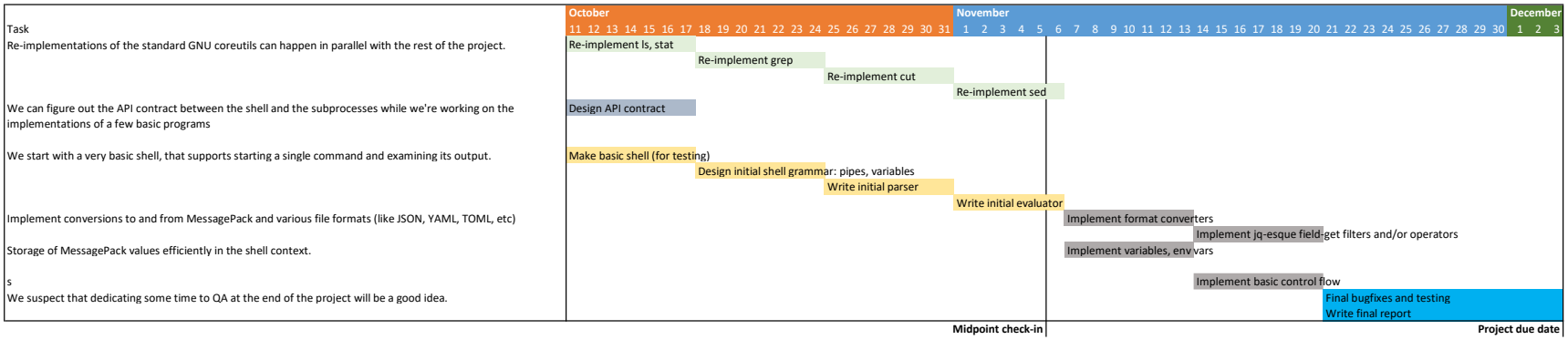
3.3 jq

`jq` is an incredibly useful shell utility used to manipulate streams of JSON documents. It's a must for any shell program that has to deal with JSON output, like the results of a JSON web API, and can be found packaged for almost every Linux distribution.[2]

JQ's principal innovation is its filter syntax. Given a JSON array of several objects, a JQ expression like `.[].things.nested_array[].value` will return a flattened list of each `value` field inside the contents of each `nested_array` found in the input stream. This kind of syntax, which implicitly maps over streams of objects, is concise and easy-to-understand, even when you need to do complex manipulations.

Monch will aim to copy much of `jq`'s syntax, and integrate it directly into the shell, instead of as a separate program. This ought to perform better, avoiding the overhead of launching an entire process to pull off a single field from an object. It also should be a lot more concise, and cleaner to use when it's integrated better into the shell.

4 Work Plan


October
November
December
11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 1 2 3
Task
Re-implementations of the standard GNU coreutils can happen in parallel with the rest of the project.
Re-implement ls, stat
Re-implement grep
Re-implement cut
Re-implement sed
We can figure out the API contract between the shell and the subprocesses while we're working on the implementations of a few basic programs
Design API contract
We start with a very basic shell, that supports starting a single command and examining its output.
Make basic shell (for testing)
Design initial shell grammar: pipes, variables
Write initial parser
Write initial evaluator
Implement conversions to and from MessagePack and various file formats (like JSON, YAML, TOML, etc)
Implement format converters
Implement jq-esque field-get filters and/or operators
Storage of MessagePack values efficiently in the shell context.
Implement variables, env vars
s
Implement basic control flow
We suspect that dedicating some time to QA at the end of the project will be a good idea.
Final bugfixes and testing
Write final report
Midpoint check-in
Project due date

5 Potential Blockers

There are several parts of this project that have the potential to be non-trivial to implement.

1. It might be too much work re-implementing enough of the coreutils to make for a useful shell environment.
2. It might be too complicated designing and parsing a syntax that is close enough to POSIX sh, but adds the extra features we want.
3. Figuring out exactly how this will be compatible with other programs might be difficult. As far as we can tell, there's no easy way to tell whether a program supports structured output without starting it and looking at what it prints. We could go down this route—guessing the type of each program's output—or just stick to a convention, like prefixing the names of all compatible programs with monch.
4. It may be difficult to support many different kinds of input and output format converters. However, this will probably be made much easier by leaning on package ecosystems like crates.io.

Bibliography

[1] Andy Chu. *Why Create a New Unix Shell?* 2021. URL: https://www.oilshell.org/blog/2021/01/why-a-new-shell.html.

[2] *jq. jq is a lightweight and flexible command-line JSON processor*. URL: https://stedolan.github.io/jq/.

[3] Microsoft. *PowerShell Documentation. Official product documentation for PowerShell*. URL: https://docs.microsoft.com/en-us/powershell/.

[4] Microsoft. *What is PowerShell? An introduction to the PowerShell scripting environment and its features*. URL: https://docs.microsoft.com/en-us/powershell/scripting/overview.

[5] Microsoft. *Working with Registry Keys*. URL: https://docs.microsoft.com/en-us/powershell/scripting/samples/working-with-registry-keys.

[6] *Oil*. URL: https://www.oilshell.org/.