Research Proposal - Object Oriented Shells

DeDominic, Anthony
Eastern Connecticut State University
Willimantic, USA
dedominica@my.easternct.edu

Abstract

The purpose of this document is to layout my plan to build my own shell which will attempt to offer a new spin to an old tool. It will dicuss the limitations in current shells for handling structured data. Second section describes the process of creating a new language as whole, defining grammars and so on. I will then layout why I believe I am qualified to see this project to completion based on my previous experiences. At the end I'll explain what I expect to create and what problems will be solvable with it.

1. Background

Shells are the primary interface into a POSIX system. As a result, many tools and utilities rely on shells to handle and process files and their data. However, shells have one strong limitation in them; the data they work with is not strongly structured. As a result, sysadmins and programmers make use of confusing and massive awk scripts and shell pipelines to derive valuable information or to make decisions. Even variables like $IFS can completely alter how these byte[] streams are handled.

If any changes were to occur in coreutils, it could break numerous tools and scripts. Even using different coreutils can lead to issues. For instance, many of GNU's coreutils, such as their implementation of grep and df, add features not strictly standardized by POSIX. Different Operating systems, ranging from GNU-based Linux distributions to non-free systems like macOS and OS X use different coreutils. Thus, this means that shells scripts can depend on behaviors and unstructured output that aren't portable.

The modern web and service oriented architectures are built on the transactions of structured documents like JSON. Currently, shells do not offer performant or efficient ways of handling these data structures. For instance, every time one would need to pull a value from a json object, it would have to be parsed by a shell tool like jq, each time. However many other scripting languages like python, perl and javascript offer such facilities For the shell to have utility in a modern cloud oriented future, it should be built to natively handle this data.

1.1. Other Attempts

An OCaml library called shcaml added a functional composer called shtreams which allows for transforming byte streams into structured documents (Heller and Tov 2008). Some developers have tried to add other structures to shell input, such as Scheme Shell and other implementations (Ellis 1980). They aimed to bring S-Expression syntax to shells. Ideally to add stronger functional properties and variadic expressiveness. It also helps with meta-programming, since it allows one to develop Domain specific languages using high order functional composition. At the end of the day, both attempts introduced systemic performance issues or failed to solve core problems, such as parsing general POSIX tools.

An older attempt to address some of these problems was using file system abstractions to mimic object oriented principles (Haemer 1994). They would spawn named pipes and shell scripts to mimic sending and retrieving values and results. This ultimately missed the point since it still relies on the same lack of native data structure support. It also had massive performance issues, since misused named pipes are slow.

1.2. Objective

Ultimately I want to take what has been learned and make a completely modern shell. A shell that has the concepts of clean, true objects. A shell that has mechanisms to structure data streams so they can be easily accessed using simple dot notation. A shell that supports modern data structures like hash maps and structured documents like JSON, YAML, XML, TOML and various others. For all features and goals, see section 4

2. Methodology

The goal is to build, not only a new shell, but basically an entire language. To do this is a multilevel step. In most cases it's usually a process of:

- building a lexical analyser
- a parser which generates an abstract syntax tree
- some kind of interpreter or compiler or virtual machine to execute that abstract syntax tree.

2.1. Lexical Analysis

This is generally the first step of building a new language. The key of this step is to take a documents and to add a token identifier to split up, individual strings. Generally errors aren't caught here, other than invalid identifier names.

2.2. Grammars (and Actions)

The key of this step is to take emitted tokens from the lexical analyser and to check them against a set of rules. Depending on the parser style depends on the kind of and quality of errors. In many cases, most users use domain specific languages like GNU bison or YACC to generate the logic behind their grammars. Without such a DSL, developers are forced to make recursive decent parser or create parser combinators (Jonnalagedda, Coppey, and et al. 2014).

The second phase of this is to apply grammars to actions. Actions are basically the logic to apply when a grammar matches. In this case, the actions could be interpreter actions or in most cases, building an abstract syntax tree.

2.3. Abstract Syntax Tree

For the sake of simplicity, optimizations will not take place. This is the area where optimizers will run, such as null code path trimming.

2.4. Interpreter

Due to the dynamic nature of a shell, it only make sense that the new language be interpreted. Interpreters are generally slower than compiled, or just-in-time compiled. Since shells are usually line-by-line oriented, it's a challenge to have a dynamic, JIT compiler; by the time it kicks in, it would be too late and the next instruction would have been loaded.

2.4.1. Planning & Implementation

General software engineering practices like diagramming should be employed to ensure a clear and concise understanding and modelling of the system. Another helpful task is to clearly state the features expected and a general, high level description of how they should work.

2.5. Timeline

- 2016-10-22: Finish all grammars for the new shell language.
- 2016-10-28: build lexical analyser, parser and generate abstract syntax tree of the language.
- 2016-10-29: Finish presentation of above works.
- 2016-11-04: Execution of ls command showing basic features.
- 2016-11-18: Most features for an ls invocation.
- 2016-11-20: presentation should be complete.
- til 2017: polish or add more features. make it complete and finish a comphrensive write up of what I learned and what was discovered.

3. Qualifications

I have a ton of experience using and writing shell scripts, both professionally and for fun. These scripts range from generic information gathering to the extremely inane, such as a web server written using bash. through these experiences I've come to appreciate what a shell is for and it's many shortfalls. With my independent studies into Domain Specific Languages, I lerned a lot about the process of designing a langauge. On numerous occasions, I've had to deal with structured data and parsing unstructured data.

More Formally, I've taken a class in compuer languages in pursuit of my undergraduate degree. In that class I had to work with tools like flex and bison; these are common tools for making computer languages. I also had to learn about the principles of designing context free and regular langauge grammars. At Cigna, I had to make use of POSIX utilities, like vmstat, to monitor servers. To make the information useful it had to be parsed into structued forms. I also had to learn to work with tools like jq, while working; it is a tool to pull data from json documents in a shell environment.

I feel with all these experiences I am more than capable of building a simple language that will mimick most shells at the minimum.

4. Expected Outcome

The Expected outcome is a fully functioning shell which supports powerful and modern data structures and objects. Features I hope to have:

- shell that can execute processes
- supports basic shell IPC like pipes
- adds extra features like event handlers
- json -> native hashmap
- native hashmap -> json
- library framework to allow devs to make there own parsers for POSIX tools.
- inline javascript using javascript template literals.
- HEREDOCS using javascript template literals

Combined with this, The project should result in a article and presentation which describes in great detail how the shell works, what it solves and other findings.

Citations

Ellis, John. 1980. “A LISP SHELL.” ACM. doi:10.1145/947639.947642.

Haemer, Jeffrey. 1994. “A New Object-Oriented Programming Language: Sh.” USENIX. https://www.usenix.org/legacy/publications/library/proceedings/bos94/full_papers/haemer.ps.

Heller, Alec, and Jesse Tov. 2008. “Caml-Shcaml: An OCaml Library for UNIX Shell Programming.” ACM. doi:10.1145/1411304.1411316.

Jonnalagedda, Manohar, Thierry Coppey, and et al. 2014. “Staged Parser Combinators for Efficient Data Processing.” ACM OOPSLA. doi:10.1145/2660193.2660241.