jcom: A JavaScript Object Shell

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

Abstract

To be written. This is a rough draft, to see the latest revision, visit https://github.com/adedomin/CSC450-project-written under the final/ directory.

1. Introduction

Currently popular shells have no built-in feature complete way of transacting or using structured documents. As a result, working with web oriented services becomes a hassle; A hassle that involves very long and contrived text pipelines full of tools like: sed, grep, awk, xargs, tr, paste, cut, tee, and so on. What results is some sort of string tokenized parser. An example of the problem can be seen below[1].

```
# gets an html webpage from duckduckgo,
# processes it into
# "url - page description" lines
# removes ad links
# and only preserves the first three
curl ${SEARCH_ENGINE}$(rawurlencode ${4}) | \
html2 | \
grep -A 2 "@class=result__a" | \
sed '/^--$/d' | \
sed '/@class/d' | \
grep -Po '(?<=\/a(=|\/)).*' | \
paste -d " " - - | \
sed 's/\(@href\|b\)=//g' | \
sed '/r\.search\.yahoo\.com/d' | \
head -n 3
```

1.1. Related Works

Object Oriented shells is not an entirely new concept. Many projects, some more serious than others, attempted to solve this issue [1] [2].

[1]source: https://github.com/GeneralUnRest/neo8ball-irc/blob/master/lib/search.sh

The power of shells is well understood. Most programming languages have some kind of system() or shell invocation mechanism. Some, such as the developers of shcaml attempted to take it further by including functional combinators; these combinators allowed for slipping in native ocaml code and objects. As a result these code pieces could be parsers, that could take known shell commands, and structure their data into key-value trees [1].

Purely object oriented attempts ultimately result in new instances of bash and explicit message passing with named pipes [2]. Even though such a system enables shells to have powerful object-like abstractions, it generally comes with significant overhead. It also incurs security risks since processes can write and listen to the objects' named pipes.

Lisp shells have been ideas attempted in the past; examples are like Scheme Shell, and other toy, academic ones [1][3]. The advantage lisp gives over other languages, which makes it suitable for shells, is that the program and data have the same form [3]. This comes in handy in meta programming, where programs can dynamically change to handle certain data. In terms of structural data, strong functional composition and combination can give structural data parsing functionality; of course at great cost of readability and complexity.

Projects like Powershell are built on the .NET runtime and can thus utilize class like features and functions. The streaming pipelines in Powershell are merely .NET classes which allow for some structural correctness and parsing [1][4]. As a result, Microsoft prefers to call more than just a shell, but a whole "automation and configuration management framework." One that is capable of handling structured data such as JSON, XML, CSV, etc, REST APIs, and object models.

Tools like Ansible, are not quite shells, but are domain specific languages which solve similar problems. Ansible and Salt Stack use code generation–and a structured document playbook file, to construct python code. Since python is a general purpose language, it handles structured and typed data much better than

shell code. As a result, powerful modules for system administration, configuration and interaction with web services have made it a formidable automation tool.

1.1. Service-oriented Architecture

Service-oriented Architectures, or SOA, are modern way to construct, highly available, data rich web applications. In short, a service oriented architecture is a way for applications to share state and provide services over middleware and structured, serialized objects. [5] One way of transacting states between services is through Representational state transfer, or REST. With REST, services are able to share well structured data through document structures like JSON, XML, etc. [6]

POSIX-shells are completely out of this domain because of their limitations. Generally when dealing with web APIs it is recommended to build a general purpose language script instead.

1.2. Designing and Defining a Domain Languages

Designing domain specific languages is a very slow and arduous task. Some have suggested designing and developing domain specific languages using meta-models; basically a model, modeling models [7]. This process makes use of advanced features of UML version 2.1. This also has the added benefit of code generation.

Defining a language is ultimately the process of creating formal grammars. Generally this is done empirically, as in from experience and trial and error. Some have used machine learning and bottom-up CYK based parsers to generate grammar [8]. This would be useful for languages built using example strings.

1.3. Events

Some system events may impact other systems. Currently, other than using "triggers", there is no way to define listen events in POSIX shells. Powershell offers such functionality through the power of its .NET classes [4].

Currently to accomplish a similar function, one must use a blocking statement like this:

```
# file descriptor is needed
# for performance reasons
exec <>99 /some/named-pipe
# consumes a process, must be subshelled.
while read -r line; do echo $line; done <&99 &
echo "event" >&99
```

Given that package management tools used in many linux distributions use shell like scripting, and have a post-install process, it may be helpful for a user to define his own post-install event handlers or notifiers.

2. 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 the GNU coreutils–such as their implementation of grep and df, add features not strictly standardized by POSIX. An example:

```
grep -P '(?>=)somepat' file
# does not work on macOS, which uses,
# mostly, BSD coreutils
```

Different operating systems–ranging from GNU-based Linux distributions, The BSDs and other non-free systems like macOS, Solaris, AIX, and other UNIX, can potentially use different coreutils. Thus, this means that shells scripts can depend on behaviors and unstructured output that are not 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. Many other scripting languages like python, perl and javascript offer such facilities; thus it should be expected that a shell should have the feature as well. For the shell to have utility in a modern cloud oriented future, it should be able to handle this data in a natural and clean way.

2.1. Other Attempts

An OCaml library called shcaml added a functional composer called shtreams which allows for transforming byte streams into structured documents [1]. Some

developers have tried to add other structures to shell input, such as Scheme Shell and other implementations [3]. 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 [2]. 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 improperly used named pipes are slow.

2.2. 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

2.3. Methodology

In order to make use of the technologies described, first the user must have NodeJS. The version of node being used on the test machine will be the latest as of this writing, v7.1.0. With NodeJS comes crucial tools like npm. npm is the tool that handles dependencies; The project makes use of such dependencies like pegjs. When downloading the source of the project from github. It is necessary to run npm install to require dependencies.

The project is version controlled using git and is hosted on a remote repository on github[2]. To check out the project, it would be advised to have a git client. npm can also download the latest build of the project giving it the github link.

[2]Source: https://github.com/adedomin/jcom

2.3.1. Libraries

The project makes use of the following libraries. For a more up-to-date list, please consult the package.json file in the project which lists all the dependencies and their semver version number:

- pegjs - parser generator
- lodash - utility functions for functional array and object handling.
- js-yaml - a yaml parser for javascript, for an example plugin
- various NodeJS built-ins: fs, process, readline, etc.

2.3.2. Testing Envrionment

The machine being used for testing is a Fedora GNU/Linux, release 24, virtual machine running on a Windows 10 Hyper-V hypervisor. This hypervisor has a Core i7-3930k processor and 32GB of memory. The machine has 2 cores of the 6 physical, reserved for itself, with dynamically resizing memory, up to 16GB. To ensure support for the latest features and capabilities, the latest release–as of this writing, v7.1.0, was installed on the machine.

3. Results

3.1. Objects and Structured Documents

The main object of the project is to process structured documents. To accomplish this, the language will offer a plugin system. Plugins in this context are merely javascript functions which return–or return a value via a callback, javascript objects.

```
// store some input or output buffer
// parsed by a plugin named parseJson
SHELL_VARS[ident] = parseJson(buffer)
// a plugin that uses xpath to select some
// values from an XML document and stores it
SHELL_VARS[ident2] = xpathSelector(buffer, args)
```

Ultimately, the variables are stored in one giant associative arrays. This is a common practice for storing variables. Python objects use dictionaries to keep track of their variables. This simplifies the management of the system. The only limitation is that it can only store un-typed javascript objects, such as Strings, Numbers, Objects and associative arrays (objects without functions).

Accessing Variables are accessed using javascript notations encapsulated in template literal-like syntax. This integrates well with the Javascript virtual machine and is likely cleaner than bash or other shells.

```
// getting an array element from an object
${somevar.somearr[1]}
// bash does not have nested objects
${somevar[1]}
// iterate over every element in an array
${somevar.forEach(var => do something)}
// bash
for var in ${somevar[@]}; do something; done
// NESTED OBJECTS
// not possible in bash as of v4.3
${somevar.someotherobj.somekey.moreobjs.etc}
```

Again, as shown in {%SOME FIGURE%} javascript notation and variable storage opens up an avenue to handle various kinds of data. This opens up possibilities to accomplish more general tasks that normally is left to general scripting languages like perl, python, ruby or even javascript with nodejs.

3.2. Events

One feature shells have is the ability to create file descriptors. With these, users have a fast way of connecting programs. File descriptors is usually how one would recreate–what could best be described, as an event loop.

Javascript offers an object called an EventEmitter. With it, one can create low coupled software triggered by events. EventEmitters are very simple, the only two methods that are of importance are on() and emit(). emit() lets one create an event labeled by a string, to be handled by an on(). The on() handles events of a particular label with a function.

In the shell, there exists a simple plugin which leverages this power. The shell would provide a builtin command, *emit* and *on*. The *on* command will take a label and a command as arguments. When executed, the system will listen for the label provided.

As a result, when a user calls an *emit* with the correct label, the *on* will be triggered and the command associated with the on will be ran. Information can be fed through the *emit* that will flow to the *on* using piping.

{% put cool graph here %}

```
# prints what it is fed
on 'echo' cat
# will print hello, world
echo 'hello, ' | emit 'echo'
echo 'world' | emit 'echo'

# reload config on config change
on 'configChange' \
    "kill -SIGWINCH ${some_server_pid}"
# load inf parser plugin
load parse-ini
parse-ini SERVER_CONFIG /etc/server_config
echo "db://newdb.url" \
    | toVar "SERVER_CONFIG.db_url"
parse-ini SERVER_CONFIG \
    --serialize -o /etc/server_config
# reload changed config
emit "configChange"
```

As shown above, these events can be incredibly useful for tasks such as reloading a server when a configuration file is changed. It also demonstrates the power of plugins system. Users can create plugins to parse their configurations into javascript objects. This allows for an easy way to manipulate them and reconstitute them, with changes, back to a file.

{% describe how it works, nodejs EventEmitter, etc. {% can do better %} %}

3.3. Command and File Parsers

As shown above, the shell can simplify some very common configuration tasks. As suggested, many common shell tasks involve handling file data. To do simple tasks like modifying configuration files can result in humorous stream processing scripts. Similar to how json objects have defined schema, many times configuration files have some formal structure. A common structure is the key:value pattern. Usually files have keys and their values and they are separated by a simple equality sign; such files are easily handled by tools like grep or sed.

More advanced configuration structures like ini files, give the user the ability to have single-level nested objects and comments. These files can still be easily transformed with basic line oriented tools. However, consider rich configuration structures like yaml or even Java .properties files. These structures can have infinitely nested objects, arrays and various primitive types. Both of these configuration files have type affinity systems. These become slightly harder to modify with line oriented tools.

By implementing parsers in javascript, a user could then load it into the shell and begin manipulating it using simple object dot notation.

```
# parse nagios's main configuration
load nagios-parser

nagios-parser --to-var NAGIOS /etc/nagios/nagios.cfg
echo ${NAGIOS.host1} # prints info related to host1
${NAGIOS.host1 = {new obj}} # new host1
nagios-parser --serialize -o /etc/nagios/nagios.cfg
```

All that needs to be done is to make a parser for it.

Plugins, in a way, are no different from normal shell programs.

3.4. Further Examples

- compare and contrast to various other tools and shells.
- benchmarking vs other tools/technologies.

3.5 Final Thought

- final word, compare to other shells and systems manipulation technologies.
- present benchmarking results.
  - graphs
  - tables

8. Discussion

- comments about the findings
  - example syntax, comparison
  - performance?
  - readability?
  - suitability over other means?
- implications of the project.
  - replace certain domain specific languages like ansible.
  - fully free and open source powershell like shell for POSIX.
  - built on a common, hackable, language many know.
- state limitations, weaknesses in product.
- further work needed.

Citations

[1] A. Heller and J. Tov, "Caml-Shcaml: An OCaml Library for UNIX Shell Programming," 2008. DOI: 10.1145/1411304.1411316

[2] J. Haemer, "A New Object-Oriented Programming Language: sh," 1994 [Online]. Available: http://https://www.usenix.org/legacy/publications/library/proceedings/bos94/full_papers/haemer.ps

[3] J. Ellis, "A LISP SHELL," 1980. DOI: 10.1145/947639.947642

[4] J. Snover, "Monad Manifesto." Microsoft, Aug-2002 [Online]. Available: http://www.jsnover.com/Docs/MonadManifesto.pdf

[5] Perrey and Lycett, "Service-oriented architecture." IEEE, Jan-2003. DOI: 10.1109/SAINTW.2003.1210138

[6] R. Battle and E. Benson, "Bridging the semantic Web and Web 2.0 with representational state transfer (REST)." Elsevier, 2008. DOI: 10.1016/j.websem.2007.11.002

[7] B. Selic, "A Systematic Approach to Domain-Specific Language Design Using UML." IEEE, May-2007. DOI: 10.1109/ISORC.2007.10

[8] K. Nakamura and T. Ishiwata, "Synthesizing Context Free Grammars from Sample Strings Based on Inductive CYK Algorithm." Springer, Grammatical Inference: Algorithms and Applications, 2000. DOI: 10.1007/978-3-540-45257-7_15