# jcom: A JavaScript Object Shell

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

## Abstract

*to be written*

## 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 isn't an entirely new concept. As others have shown, many projects, some more serious than others, attempted to solve this issue [@shcaml] [@oosh].

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 [@shcaml].

Purely object oriented attempts ultimately result in new instances of bash and explicit message passing with named pipes [@oosh]. Even though offers powerful object-like abstractions it general comes with significant overhead. It also incurs security risks since processes can write and listen to the object named pipes.

Lisp shells have been ideas attempted in the past; examples are like Scheme Shell, and other toy, academic ones [@shcaml][@lispsh]. The advantage lisp gives over other languages, which makes it suitable for shells, is that the program and data have the same form [@lispsh]. 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 [@shcaml][@monadshell]. 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, aren't quite shells, but are domain specific languages which solve similar problems. Ansible, et al, use code generation and a yaml 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] source: https://github.com/GeneralUnRest/neo8ball-irc/blob/master/lib/search.sh

### 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. [@soa] 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. [@semanticrest]

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 arduous problem. Some have suggested designing and developing domain specific languages using meta-models; basically a model, modeling models [@dsldes]. 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 [@synthcfg]. 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 [@monadshell].

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 GNU's 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 BSD coreutils
```

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 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. 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.

### 2.1. Other Attempts

An OCaml library called shcaml added a functional composer called shtreams which allows for transforming byte streams into structured documents [@shcaml]. Some developers have tried to add other structures to shell input, such as Scheme Shell and other implementations [@lispsh]. 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 [@oosh]. 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

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, compiler or virtual machine to execute that abstract syntax trees.

### 2.4. Materials

- list of technologies.
  - javascript
  - nodejs
- libraries.
  - pegjs
- machine used in benchmarking/experiment.

# 3. Objects and Structured Documents (part of methods)

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 a 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.

# 4. Events (part of methods)

- feature showing off *emit* and *on* keyword.
- describe how it works, nodejs EventEmitter, etc.
- use case examples.
  - show use cases.

# 5. Command and File Parsers (part of methods?)

- section to show off feature.

- describe how it allows one to create parser modules for commands and common files.
- examples of uses.
  - ideally just a few commands, like ls, etc.
  - show how it can be used to easily configure some program.
  - nagios, apache, something.

## 6. Further Examples

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

## 7. Results | Findings

- 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