The POSIX shell is an interactive DSL for concurrency

Michael Greenberg[*]
Pomona College
michael@cs.pomona.edu

1 Introduction

The POSIX shell is the *de facto* standard for interacting with computer systems, ranging from personal laptops to clusters of many powerful servers [IEEE and The Open Group 2016]. Few languages rival the shell for power, but it has inspired more mockery and revulsion [Garfinkel et al. 1994] than academic attention [D'Antoni et al. 2016; Jeannerod et al. 2017a,b; Mazurak and Zdancewic 2007]. The shell is not taken seriously as a programming language, but it is in fact an extremely powerful DSL for concurrency with strong support for interactivity. In my talk, I will present preliminary work on formalizing the POSIX shell standard. What makes the POSIX shell so good at interactivity [Greenberg 2018] and managing concurrency [Greenberg 2017]? How can we help novices and more experienced users understand the POSIX shell? I will demonstrate a stepper that makes the shell's obscure semantics observable.

2 The shell is an interactive DSL

While the shell is used to power complex systems (e.g., Debian maintainer scripts [Jeannerod et al. 2017b]), it also sees widespread use as an interactive console—the expert's control hatch. On many systems, there are management tasks that can *only* be done via the shell. But even when GUI options exist, many users prefer the shell. What makes the shell so good for interactivity?

I believe there are two aspects of the shell's design that support interactivity: like many other DSLs, the POSIX shell offers extremely concise syntax for common commands (e.g., `mv *.c src/`); and many of the commonly used shell commands are *variadic*. For example, the `mv` command above has a flexible interface, of the form `mv` $\texttt{file}_1$ `...` $\texttt{file}_n$ `tgt`, where `tgt` must be a directory if $n > 1$ but need not be if $n = 1$. The `mv` command's interface is directly supported by the shell's *expansion semantics*. In the shell, commands and their arguments are treated 'stringily': at runtime, a process called *expansion* translates strings containing control codes (like the `*` in the command above, or variables like `$PATH`) to possibly many arguments. The expansion of `*.c`, for example, will search the current working directory for files that match (i.e., end in `.c`), generating and alphabetically sorting an argument for each such file.

[*]Written while a visiting scholar at Harvard University.

DSLDI'18, 2018-11-06, SPLASH Boston, MA, USA
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00

Greenberg [2018] has already argued that expansion supports POSIX shell interactivity: expansion dovetails nicely with variadic commands and is also well suited to the core tasks of the shell (specifying executable names, arguments, and paths). For more information on how word expansion, see either Section 2 of that paper or the POSIX standard [IEEE and The Open Group 2016].

3 The shell is a DSL for concurrency

The POSIX shell succinctly and powerfully manages the flow of information between concurrent processes. It has many primitives for managing file descriptors (>, <&, etc.), setting up pipes between processes (`|`, command substitution, etc.), and controlling concurrent jobs (&, `wait`, etc.). Each command in the shell runs in a separate memory space, using the filesystem as shared memory. To see how simple it is to work with concurrency in the shell, contrast the following two snippets that run LDA [Blei et al. 2003] to generate topic models for moderately large datasets [Greenberg et al. 2015]:

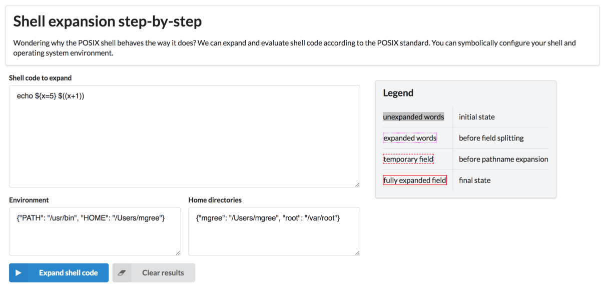
```
for k in ${KS}; do
  lda est 1/50 ${k} \
      settings.txt ${ABS} \
      seeded ${DIR}/lda${k}
done
echo "DONE"
```

(a) Sequential

```
for k in ${KS}; do
  lda est 1/50 ${k} \
      settings.txt ${ABS} \
      seeded ${DIR}/lda${k} &
done
wait
echo "DONE"
```

(b) Concurrent and parallel

In the sequential version (a), we run `lda` for each value of the parameter k sequentially. When each LDA process completes some hours or days later, it will return control to shell, which will continue the `for` loop. When the for loop terminates, we announce our completion (`echo`). Each run of `lda` is independent—we can speed things up by building models concurrently. The concurrent version (b) makes only two changes. First, we run `lda` in the 'background' (by adding & to the command). Commands run in the background return control to the shell immediately: the `for` loop will spawn all of the `lda` processes in the same order—but without waiting.



Shell expansion step-by-step

Wondering why the POSIX shell behaves the way it does? We can expand and evaluate shell code according to the POSIX standard. You can symbolically configure your shell and operating system environment.

Shell code to expand

```
echo ${x=5} $((x+1))
```

Legend

unexpanded words	initial state
expanded words	before field splitting
temporary field	before pathname expansion
fully expanded field	final state

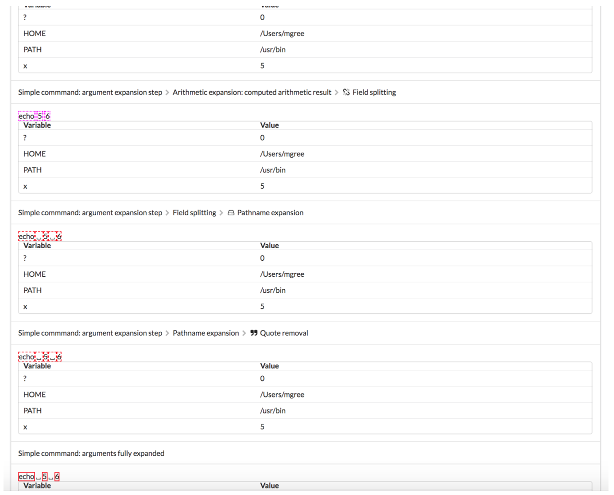
Environment

```
{"PATH": "/usr/bin", "HOME": "/Users/mgree"}
```

Home directories

```
{"mgree": "/Users/mgree", "root": "/var/root"}
```

(a) Entering a program to step



Variable	Value
?	0
HOME	/Users/mgree
PATH	/usr/bin
x	5

Simple commmand: argument expansion step > Arithmetic expansion: computed arithmetic result > Field splitting

```
echo 5 6
```

Variable	Value
?	0
HOME	/Users/mgree
PATH	/usr/bin
x	5

Simple commmand: argument expansion step > Field splitting > Pathname expansion

```
echo␣5␣6
```

Variable	Value
?	0
HOME	/Users/mgree
PATH	/usr/bin
x	5

Simple commmand: argument expansion step > Pathname expansion > Quote removal

```
echo␣5␣6
```

Variable	Value
?	0
HOME	/Users/mgree
PATH	/usr/bin
x	5

Simple commmand: arguments fully expanded

```
echo␣5␣6
```

Variable	Value

(b) Results of stepping, highlighting field separators

Figure 1. The POSIX shell stepper in action

We then use the built-in `wait` utility to have the shell wait for all of its background processes to terminate. In concurrency terminology, we've just used fork/join parallelism: the `&` is a "fork" and `wait` is a "join". The speedup from building the models in parallel and concurrently on a multicore machine are appreciable—nearly linear. Beyond the speedup, parallelizing the sequential code is simple—the Levenshtein distance between the two snippets is 7.

4 A stepper for the shell

I have built a web-based stepper that uses a mechanized semantics of the POSIX shell to visualize the various stages of expansion and evaluation (interface samples in Figure 1). It is a from-scratch small-step operational semantics written in the Lem language [Mulligan et al. 2014]. The implementation uses type classes to abstract over operating system features like reading and writing to file descriptors or calls to `execve`: the stepper uses a symbolic implementation of OS features; a separate concrete implementation makes real system calls and yields a working shell with identical logic to the symbolic one.

Even experienced shell users can be confused by the POSIX shell's semantics, particularly by *field splitting*, a late step in expansion that breaks apart the result of earlier stages into a command's arguments. Mazurak and Zdancewic [2007]'s analysis focused on finding errors where field splitting produces too many or too few fields. Greenberg [2018] gives an example showing that "field splitting can be controlled at use sites but not at definition sites" (Section 2). Here is another example of field splitting's trickiness: suppose we are in a directory with two files, `a` and `b` and we want to move these files to another directory, `../other_dir`. In typing out the command `mv * ../other_dir`, our finger slips and we hit enter before we type the target directory, running the command `mv *`. What happens? An intuitive mental model of the shell might expect an error: after all, `*` means "all files" and we never specified a destination. Without a destination, `mv` doesn't know what to do, and should produce an error. Unfortunately, that's not what happens. In fact, the shell expands `*` to the two files present, `a` and `b`, in that order. Field splitting produces two arguments to `mv`: first `a` and then `b`. That is, we run `mv a b`, overwriting `b` irrevocably with the contents of `a`.

My stepper will allow both novices and experienced shell users to observe and understand the shell's expansion behavior. With some more work, it could also do a good job showing the concurrent aspects. (The current rendering engine treats subshells and background processes opaquely, but this is a property of the JavaScript display, not operational semantics themselves.) I conjecture it will be a useful teaching tool. I plan to demonstrate the tool at the workshop and hope to receive feedback.

Acknowledgments

Austin J. Blatt worked on many parts of the stepper and helped inform my thoughts on the shell.

References

David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. Latent Dirichlet Allocation. *J. Mach. Learn. Res.* 3 (March 2003), 993–1022. http://dl.acm.org/citation.cfm?id=944919.944937

Loris D'Antoni, Rishabh Singh, and Michael Vaughn. 2016. NoFAQ: Synthesizing Command Repairs from Examples. *CoRR* abs/1608.08219 (2016). http://arxiv.org/abs/1608.08219

Simson Garfinkel, Daniel Weise (Ed.), and Steven Strassman (Ed.). 1994. *The UNIX Hater's Handbook*. IDG Books Worldwide, Inc., San Mateo, CA, USA.

Michael Greenberg. 2017. Understanding the POSIX Shell as a Programming Language. OBT.

Michael Greenberg. 2018. Word expansion supports POSIX shell interactivity. In *Programming Companion (presented at Programming eXperience (PX))*. ACM. https://doi.org/10.1145/3191697.3214336

Michael Greenberg, Kathleen Fisher, and David Walker. 2015. Tracking the Flow of Ideas through the Programming Languages Literature. In *SNAPL (Leibniz International Proceedings in Informatics (LIPIcs))*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.), Vol. 32. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 140–155. https://doi.org/10.4230/LIPIcs.SNAPL.2015.140

IEEE and The Open Group. 2016. *The Open Group Base Specifications Issue 7 (IEEE Std 1003.1-2008)*. IEEE and The Open Group.

Nicolas Jeannerod, Claude Marché, and Ralf Treinen. 2017a. A Formally Verified Interpreter for a Shell-Like Programming Language. In *Verified Software. Theories, Tools, and Experiments - 9th International Conference, VSTTE 2017, Heidelberg, Germany, July 22-23, 2017, Revised Selected Papers*. 1–18. https://doi.org/10.1007/978-3-319-72308-2_1

Nicolas Jeannerod, Yann Régis-Gianas, and Ralf Treinen. 2017b. *Having Fun With 31.521 Shell Scripts*. Technical Report hal-01513750.

Karl Mazurak and Steve Zdancewic. 2007. ABASH: Finding Bugs in Bash Scripts. In *PLAS*. 105–114. https://doi.org/10.1145/1255329.1255347

Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. 2014. Lem: Reusable Engineering of Real-world Semantics. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, USA, 175–188. https://doi.org/10.1145/2628136.2628143