# 1 Introduction

Expert computer users interact with the computer using textual interfaces, also called *command-line interfaces* or *shells*. Experts prefer shells for their concision and power. Critical system tasks—installation and deployment, automation of routine tasks, maintenance, forensics—are often done via the shell; some tasks can *only* be done via the shell. Shells are synonymous with complete control. Code for completely compromising a computer system is called "shellcode": once an attacker has privileged shell access, they control the system entirely. The shell is used on a sliding scale of interactivity: users can control their computer command-by-command, while administrators and programmers write immensely powerful shell scripts: a single script can create, modify, delete, and move files not just on a local computer, but on widely distributed servers.

Among shells, the POSIX shell is the *de facto* standard; by lines of code, it is the 9th most commonly used language on GitHub [27]. The POSIX shell standard [39] defines the most widely supported shell, specifying its syntax and semantics in 122 pages, not counting other operations that may or may not be built into the shell (e.g., `kill`, the command for sending signals to processes). When writing portable scripts, programmers look to the POSIX standard.

For all of its power and ubiquity, you don't need to look far to find examples of fragile, dangerous, or confusing shell scripts. One recent example is "Steam cleaning" [58]: a shell script in the Steam gaming platform encountered an unexpected filesystem arrangement on a user's PC... and deleted all of a user's files. Despite the untrammeled power of the shell, it's not uncommon to see software installed via `curl http://cool.io/install.sh | sudo bash`, which fetches a script from URL and feeds the script directly to a shell running as the superuser!

Developer operations, or DevOps, is a rapidly growing ($2 billion as of 2015 [63]) sub-industry, concerned with managing developer tools and deployment systems in a unified way. Many DevOps tools—Vagrant, Docker, Puppet—rely on shells scripts in one way or another. Rehearsal, a research tool that supports DevOps by modeling and verifying Puppet scripts, has to give up when a Puppet script uses shell code [68]; by not formally understanding the shell, we restrict the amount of help we can give DevOps programmers.

The POSIX shell is a widely used, powerful interactive programming language with inadequate safeguards. Shunned as a shoddy tool used only out of necessity, the shell is effectively unstudied in the programming languages community (but see Section 6 for a discussion of related work). All of the usual costs and risks apply: programming error is extremely costly, and the shell is particularly error prone compared with conventional languages! Since shell scripts are often the "glue" for DevOps, the situation is a dire one: an especially error-prone, unstudied language is used to manage production deployments of software.

This proposal presents research and education plans to mitigate these risks and costs by studying and improving the shell. I propose to study the shell for three ends: to build better tools for the POSIX shell as it exists now; to extract insights on language design for better interactive languages in general (and for the shell in particular); and to help educate students, programmers, and administrators on safe shell use.

## Contributions

I propose to study the shell as a programming language: computer users have much to gain from increased tool support for the shell; programming languages researchers have much to learn, as the POSIX shell has several distinctive features—in particular, users can slowly transition from line-by-line interactive use to interactive use with programmatic automation to full-blown programming.

My proposed plan is fourfold: first, to formalize the POSIX shell standard; second, to use

the formalization to build tool support for the POSIX shell; third, to extract language design principles from the shell for future shell-like languages and for interactive programming in general; and fourth, to develop a curriculum for safer shell programming.

Formalizing the POSIX shell standard (Section 3.1) will clarify the English of the POSIX standard; it will build a foundation for other researchers to work on the shell; and it will be a novel and interesting model of the shell's unique, expansion-based semantics. Similar efforts at formalizing, e.g., the C standard or the Java memory model have led to significant safety improvements in compilers for those languages (around undefined behavior [1] and data races [50], respectively); the language formalizations also serve as a strong foundation for the research community. We can expect similar gains for the shell. I plan to use the Lem language [57] to build the formal model: Lem models can be extracted into executable code as well as theorem provers. There will be only *one* model: the OCaml code I test against the POSIX test suite will be the same as the model I prove theorems about in Coq.

The tools I build using my formalization will be grounded in my semantics (Section 3.2). Not only will I extend existing tools, like the ShellCheck linter [45], to account for the shell's semantics; I will be able to build other tools to support shell programmers improving efficiency and by catching or preventing bugs. For example, I am already building an explorable model of string expansion for those learning the shell; I can imagine many others—compilers to general-purpose languages, to help shell scripts evolve into robust programs; code hardeners/cruft inserters to avoid common bugs in, e.g., signal handling; or analyzers for `curl`-based installation scripts. Since the Lem model will be extract to executable code, I'll be able to construct tools that directly use my validated model of the shell's semantics.

Finally, I aim to distill language design principles for future shell-like languages and for interactive programming in general (Section 3.3). As I build the semantics and tools, I will better understand which features are particularly difficult for analyses or enabling of interactivity. On the one hand, I hope to improve the shell itself, making it safer and more reliable without sacrificing its quintessential interactivity and power. On the other hand, I hope to arrive at general, foundational principles for interactive programming, beyond the shell and its particular setting.

Preliminary work has shown that PL techniques, like structural operational semantics, can formalize the POSIX standard, clarifying the standard's English with precise mathematics [30]. Early efforts at formalizing string expansion have already found a bug in `dash`'s compliance with the POSIX standard [17, 31].

# 2 Distinctive features of the POSIX shell

Three features distinguish the shell: its evaluation model, its facility for controlling concurrent processes, and its natural transition from command-at-a-time interactivity to serious programming (e.g., the Linux startup routine). We briefly describe these features here, going into more formal detail with syntax and semantics in Section 3, after which we discuss the proposed work.

## 2.1 Expansion over evaluation

The POSIX shell diverges from the ordinary evaluation model. Conventional programming languages evaluate an expression by evaluating its parts; the POSIX shell evaluates an expression by *expanding* its parts.

For example, the shell runs the expression `echo $((1 + 1)) >${f-two.txt}` as follows: recognizing a simple command, `echo $((1 + 1))`, with redirected output, `>${f-two.txt}`, the shell

```
echo "RUNNING LDA"

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


echo "DONE"
```

(a) Sequential

```
echo "RUNNING LDA"

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

Figure 1: Sequential vs. parallel shell scripts for data processing

expands the parts of the command and redirection before running the command proper. First, *arithmetic expansion* takes `1 + 1` to `2`, yielding the command, `echo`, with just the one argument, `2`. Next, the redirection is expanded: the variable reference `${f-two.txt}` means "look up `f` and return its value; if `f` is unset, use `two.txt`" as the default. Supposing that `f` is indeed unset, *parameter expansion* yields the redirection `>two.txt`, so the `echo` command's standard output (file descriptor 1) is directed to the file `two.txt`.

To see how expansion is the default, suppose we naïvely embed another command, writing: `echo $((1 + 1)) >ls two.txt`. A programmer might think that the command `ls two.txt` will be evaluated. Instead, the shell interprets `ls` as the target of the redirection—even though we intended it to be a command—and, bafflingly, parses `two.txt` as an extra argument to `echo`. After expansion, we'll have the same command—`echo`—with two arguments, `2` and `two.txt`; the output will be redirected to the file `ls`.

For the shell to evaluate the subparts of an expression, one must ask for *command substitution*, by using backticks, `` `ls two.txt` ``, or a parenthesized form `$(ls two.txt)`—single parens, rather than double as for arithmetic). When expanding `echo $((1 + 1))>`ls two.txt``, the shell runs the command `ls two.txt`, using its *output* as the target of redirection.

## 2.2 Controlling concurrent processes

The POSIX shell has many primitives for managing file descriptors (via `>`, etc.), setting up pipes between processes (via `|`, command substitution, etc.), and controlling concurrent jobs (via `&`, `wait`, etc.). To see how simple it is to work with concurrency in the shell, contrast two pieces of code (Figure 1) for running LDA to generate topic models for moderately large datasets [8, 32].

In the sequential version (Figure 1a), we tell the user the task is starting (the first `echo`), and then run `lda` for each value of the parameter $k$. When the LDA process completes, it will return control to shell, which will continue the `for` loop. The `for` loop is the conventional sequential one: we won't start the next iteration until the first one has completed. When the for loop terminates, we announce our completion (the final `echo`). A large model can take as long as two days to generate; since our script is sequential, we'll build each model in turn, one at a time. If we are building five moderately sized models, a sequential run will take about five days.

Each run of `lda` is independent, so we can run all of them concurrently. Only two changes need to be made. First, we run each lda command in the 'background' (by adding `&` to the command). Commands run in the background return to the shell immediately: so the `for` loop will quickly

```
$ grade hw1-latest
$ parse hw1-latest/grades.txt >hw1.csv
$ grade hw2-latest
$ parse hw2-latest/grades.txt >hw2.csv
...
```

(a) Manually running commands

```
for d in *-latest; do
  echo ${d} ${d%-latest}
done
```

**Output:**

```
hw1-latest hw1
hw2-latest hw2
...
```

(b) Loop that prints commands

```
for d in *-latest; do
  echo ${d%-latest}
  grade ${d}
  parse ${d}/grades.txt \
        >${d%-latest}.csv
done
```

**Output:**

```
hw1
hw2
...
```

(c) Finalized loop with diagnostic output

Figure 2: Interactive programming in the shell

spawn a number of `lda` processes all at once. We then use the built-in `wait` 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. But just as importantly, the process for parallelizing the sequential code is simple—simpler than in any 'general purpose' programming language!

Each command in the shell runs in a separate memory space, using the filesystem as shared 'memory'. In addition to its concurrency controls, the POSIX shell has succinct syntax and powerful semantics for managing the flow of information between concurrent processes: pipes allow for programs to be composed, while redirections capture information to the filesystem.

### 2.3 Interactivity, live coding, and automation

The shell is, before anything else, a way of interactively controlling a "computer's whirring innards" [20]. While many programming languages offer an interactive mode—a read-eval-print loop, or REPL—the shell is *essentially* interactive. Most shell users work only interactively; one might think that writing a shell script is a more formal process, like programming in other languages, but even the process of writing a shell script is an interactive, iterative one. The shell encourages a sliding scale from interactivity to batch programming using an iterative "print what you'll do before you do it" programming style. For example, when facing a repetitive grading task, I might observe a pattern in the commands I'm running one at a time (Figure 2a); can I automate the process? Since the shell is such a powerful tool, I'll start out with some exploratory debugging. Automatically running the wrong commands can be destructive and hard to undo! I'll start by trying my hand at writing a `for` loop, but rather than actually *executing* commands, I

simply print out some values derived from the loop variables at each iteration (Figure 2b). Once I've found the necessary expansions, it's safe to actually run the commands (Figure 2c), perhaps with some diagnostic printing to allow for a manual abort in case things go off the rails. First, we can build tools to help support this programming model by offering speculative execution or rollback facilities, extending the idea of the `maybe` tool [76]. Programming languages are seen as a means for automation, even though conventional languages often perform tasks very differently than humans would. The shell automates human tasks more literally: the `for` loop above runs exactly the commands I would have run manually. By literally "doing what a human would do", the shell is more like a macro system (in the sense of, e.g., Microsoft Office macros) than a conventional programming language. Programmers in conventional languages would have written a slightly different program (probably with explicit data structures for working with directories and files and several calls to system functions to redirect the output of the `parse` program into the CSV file) and wouldn't have used the same "print first, run later" strategy. That is, programming the shell happens at a human scale—and *interactively*.

Some of the shell's interactivity is historical—it was designed to be interactive and *became* a programming language, but I believe that some of the shell's interactivity comes from its particular constellation of features. In particular, string expansion seems to be at the center of what makes the shell so interactive. Is there an interactive programming language that is close enough to the POSIX standard that simple programs still work, but is more predictable and less error-prone? Can we study the shell programs that people actually write to find a narrower, safer set of features? Does the shell offer insights into interactive programming more generally?

# 3 Research plan

## 3.1 Formalizing the POSIX shell

The groundwork for the proposed research is a formal model of the POSIX shell. Where the English of the POSIX standard is open to interpretation, a mechanized, formal model of the shell is unambiguous. Clarifying and disambiguating the POSIX shell's English serves many stakeholders: anyone implementing, building tool support for, or programming in the shell will have a definite reference. Moreover, the shell's string expansion semantics is novel, so our formal model will also offer a new object of study for theorists.

My initial efforts at formalizing the shell have used Lem, a functional programming language designed for modeling complex systems [57]. One of Lem's key features is *extraction*: from a single model, Lem can extract both efficiently executable code in OCaml as well as definitions suitable for the Coq proof assistant [18]. Since both the code and the prover definitions came from a single source, we can be confident (inasmuch as we are confident in Lem's implementation) that any proof about our system in Coq applies to our running code in OCaml.

I plan on implementing the core POSIX standard, even though every shell goes slightly beyond the standard—dash just a bit, bash quite a lot, and fish even more so [17, 4, 21]. Most extensions would not be hard to model, but the most benefit seems to come from building a model of core functionality. These extensions will be more relevant as I think about language design (Section 3.3).

### 3.1.1 Syntax

The shell's concrete syntax is well enough documented in the POSIX standard [39]; we have defined an abstract syntax that eliminates redundancies and is amenable to modeling with a small-step operational semantics (Figure 3). Readers familiar with shell programming will note that the

**Syntax**

| | | | |
|---|---|---|---|
| Commands | $c \in \mathsf{C}$ | ::= | $: \mid \vec{a}\ \vec{w} \mid c_1 \mid c_2 \mid c\ r \mid c\ \& \mid \mathsf{sub}\ c \mid c_1; c_2 \mid$ $c_1\ \&\&\ c_2 \mid c_1 \mid\mid c_2 \mid !c \mid \mathsf{if}\ c_1\ c_2\ c_3 \mid \mathsf{case}\ w\ \vec{b} \mid$ $\mathsf{while}\ c_1\ c_2 \mid \mathsf{for}\ s\ w\ c \mid s()\ c$ |
| Redirections | $r \in \mathsf{R}$ | ::= | $\mathsf{write}\,t \mid \mathsf{writenc}\,t \mid \mathsf{read}\,t \mid \mathsf{rw}\,t \mid \mathsf{append}\,t \mid \ldots$ |
| Targets | $t \in \mathsf{Tgt}$ | ::= | $w \mid \&d$ |
| File descriptors | $d \in \mathbb{N}$ | | |
| Assignments | $a \in \mathsf{Asgn}$ | ::= | $s = w$ |
| Branches | $b \in \mathsf{B}$ | ::= | $p)\ c$ |
| Patterns | $p \in \mathsf{P}$ | ::= | $w \mid w\vert ps$ |
| Words | $w \in \mathsf{W}$ | ::= | $\vec{e}$ |
| Entries | $e \in \mathsf{E}$ | ::= | $s \mid k \mid \text{␣}$ |
| Control codes | $k \in \mathsf{K}$ | ::= | $\sim \mid \sim s \mid \$\{s\vert\phi\} \mid \$(c) \mid \$((w))$ |
| Parameter formats | $\phi \in \Phi$ | ::= | $\mathsf{normal} \mid \mathsf{length} \mid$ $\mathsf{default}\ w \mid \mathsf{ndefault}\ w \mid \mathsf{assign}\ w \mid \mathsf{nassign}\ w \mid \ldots$ |
| Strings | $s \in \Sigma^*$ | | where $\Sigma$ is, e.g., UTF-8 |

Figure 3: Abstract syntax of the POSIX shell

abstract syntax cleans up the shell's concrete syntax, no longer, e.g., terminating `case` statements with `esac`. I use sans-serif fonts for abstract syntax and `typewriter fonts` for concrete syntax.

The shell's abstract syntax comes in two parts: *commands* are the "programming language" part, with structured control; *words* are the "expansion" part, which are only lightly structured. Commands all return an *exit code*, where 0 means success and anything else means failure; words expand into strings, which are broken up into lists of strings during *field splitting*.

**Commands** The command : (pronounced "colon" or "skip") is a no-op. The *simple* command $\vec{a}\ \vec{w}$ has two parts: $\vec{a}$, the *assignment part*, is a list of assignments; $\vec{w}$, the *command part*, is a list of words. Simple commands can run actual executables, shell "built-ins", or user-defined functions. To determine what, precisely, a simple command does, we must first completely expand the words in the assignment and command parts. The first word of whatever the command part expands to determines which type of command will be run. The *pipe* command $c_1|c_2$ forwards $c_1$'s output to $c_2$'s input. The *redirection* command $c\ r$ runs the command $c$ under the redirection $r$, which might send output to a file ($\mathsf{write}\,t$; concrete syntax `>t`) or have a given file descriptor $d$ read from a given source ($d\,\mathsf{fdread}\,t$; concrete syntax `d<t`, where $d$ is a file descriptor number). The *background* command $c$ & causes $c$ to run in the background—control is immediately returned to the shell while $c$ runs. The *subshell* command $\mathsf{sub}\ c$ runs $c$ inside of a "subshell"—a copy of the current shell environment. Any effects $c$ has on the shell environment only lives as long as the subshell does. (Confusingly, the concrete syntax for subshells is parentheses, `(c)`. The shell uses curly braces, `{ c }`, for grouping commands and disambiguating precedences.) The *sequence* command $c_1; c_2$ runs $c_1$ and then $c_2$.

There are a series of *logical* commands: && (read "and"), || (read "or"), and ! (read "not"). These are "short circuiting" operators, e.g., $c_1$ && $c_2$ runs $c_1$, running $c_2$ only when $c_1$ succeeds. There are also two *conditional* commands: if and case. The if statement $\mathsf{if}\ c_1\ c_2\ c_3$ works like

conventional conditionals, using $c_1$'s exit code to select one of $c_2$ and $c_3$. The case statement case $w$ $\vec{b}$ expands the word $w$ and searches the *branches* $\vec{b}$ for the first match; each branch $b$ has a pattern $p$ (comprising one or more alternatives) and a command to run on a match.

There are two (core) *loop* commands: while and for. The while loop while $c_1$ $c_2$ runs as long as the condition $c_1$ succeeds. (The shell also has an `until` loop, which can be encoded with syntactic sugar.) The for loop for $s$ $w$ $c$ expands $w$ into some number of fields; the string $s$ serves as a variable name for iterating through each of the fields: $c$ is evaluated with $s$ bound to the first field from $w$, and with $s$ bound to the second field from $w$, and so on.

**Words** Words are the parts of the program that are subject to the shell's seven stages of *word expansion*. Words are only loosely structured: each word is a sequence of *entries*, $e$, where each entry is either an uninterpreted string, $s$, a *control code*, $k$, or a *field separator*, which we write as a visible space, ␣, and pronounce "sep".

Control codes are where expansion actually happens. Two forms of the *tilde* control code, $\sim$ and $\sim s$, expand to a user's home directory: the plain tilde expands to the current user's home directory; the tilde followed by a string $s$ expands to the home directory of the user specified by $s$. (The username $s$ must be a string, not an expandable word.) *Variable references*, written $\${s|\phi}$, lookup the variable $s$ in the environment and apply a *parameter format*, $\phi$. Parameter formats amount to tiny built-in functions: $\${x|\mathsf{normal}}$ (concrete syntax `${x}`) will expand to whatever the variable $x$ holds; $\${x|\mathsf{length}}$ (concrete syntax `${x#}`) will expand to the length of whatever string the variable $x$ holds; $\${x|\mathsf{default}\,w}$ (concrete syntax `${x-w}`) will either expand to what $x$ holds if $x$ is defined—if $x$ is undefined, it will expand to whatever $w$ expands to. The ndefault prefix has the same behavior as default prefix, except that the former chooses the default if $x$ is undefined or if $x$ is null. *Command substitutions*, written $\$(c)$, expand to the output found when running the command $c$. (Readers may be more familiar with the equivalent concrete syntax `` `c` ``.) *Arithmetic substitutions*, $\$((w))$, expand $w$ to a string and then parse the string as a variant of the ISO C standard arithmetic operators; the arithmetic substitution evaluates a string representation of the resulting number.

### 3.1.2 Semantics: expansion and evaluation

The shell's semantics are amenable to existing PL techniques—small-step operational semantics in particular. String expansion introduces a novel step of indirection: evaluation and expansion are mutually recursive processes, guided by the programmer. Most languages evaluate a term by evaluating its subparts; by default the shell works by *expanding* the subparts. To have the parts of a term evaluated, the programmer must specifically ask for it by using a command substitution.

Our evaluation relation must keep track of the state of the world: the filesystem, fs; the active processes, $P$; and the shell state, $\rho$, which encompasses the shell's environment and other important data, e.g., the working directory, signal traps, etc. We group the three of these in the tuple sys. We can define the small-step semantics as a pair of mutually recursive transition systems: we write $\mathsf{sys}, w \longrightarrow \mathsf{sys}', w'$ for string expansion, i.e, that under the system sys, the word $w$ expands to the system $\mathsf{sys}'$ and the word $w'$; we write $\mathsf{sys}, c \longrightarrow \mathsf{sys}', c'$ to mean the more conventional command evaluation, i.e., $c$ evaluates to $c'$ updating sys to $\mathsf{sys}'$.

The semantics here are a cartoon of what we have formally developed in Lem, with a few sample rules shown to give the flavor. Strictly speaking, the rules shown here are neither complete nor correct, but they ought to give the intuition. We use `typewriter fonts` to indicate system calls and other internals, which we leave unspecified here (but will need to be characterized in the formalization; see Section 3.1.3).

**Runtime terms and system configurations**

| | | | |
|---|---|---|---|
| Runtime commands | $c \in \mathsf{C}$ | ::= | $\ldots \mid \texttt{execve}(s, \vec{s}, \mathsf{env})$ |
| Runtime entries | $e \in \mathsf{E}$ | ::= | $\ldots \mid \mathsf{exp}\, s \mid \mathsf{quoted}\, s \mid \mathsf{cmd}\, c\ w$ |
| Fully expanded words | $f \in \mathsf{Exp}$ | ::= | $s \mid \text{“}\vec{f}\text{”} \mid \sqcup$ |

| | | | |
|---|---|---|---|
| System state | $\mathsf{sys} \in \mathsf{Sys}$ | ::= | $\langle \mathsf{fs}, P, \rho \rangle$ |
| Filesystem state | $\mathsf{fs} \in \mathsf{FS}$ | | |
| Process state | $P : \mathsf{PId} \rightharpoonup \mathsf{C} \times \mathsf{Env}$ | | |
| Shell state | $\rho \in \mathsf{Sh}$ | ::= | $\{\mathsf{env} \in \mathsf{Env}, \ldots\}$ |
| Environment | $\mathsf{env} \in \mathsf{Env} = \Sigma^+ \rightharpoonup \Sigma^*$ | | |
| Process ids | $p \in \mathsf{PId} = \mathbb{N}$ | | |

**Metafunctions**

| **String expansion** | | **Command evaluation** | |
|---|---|---|---|
| Field splitting | $\mathsf{str} : \mathsf{Exp} \to \overrightarrow{\Sigma^*}$ | Environment export | $\mathsf{export} : \mathsf{Env} \to \mathsf{Env}$ |
| Parse arithmetic | $\mathsf{parse} : \mathsf{Exp} \rightharpoonup \mathsf{Arith}$ | Path lookup | $\mathsf{find} : \mathsf{Env} \times \mathsf{FS} \times \Sigma^+$ |
| Evaluate arithmetic | $\mathsf{eval} : \mathsf{Env} \times \mathsf{Arith} \rightharpoonup \mathsf{Env} \times \mathbb{Z}$ | | |
| Stringify numbers | $\mathsf{itoa} : \mathbb{Z} \rightharpoonup \Sigma^+$ | | |

Figure 4: Runtime terms and metafunctions

**String expansion.** String expansion, written $\mathsf{sys}, w \longrightarrow \mathsf{sys}', w'$, takes a system configuration $\mathsf{sys}$ and a word $w$ (a/k/a a list of entries) and hunts through the word for the leftmost control code (STEP, CONTROL; the rules for control codes are of the form C*; Figure 5).

*Tilde expansion* defers to the system, either looking up the user's home directory (CHOME) or making a system call to find the given user's home directory (CUHOME). Since tilde expansion is the first stage of expansion to run, we effectively require that any user is given in advance as a fully-expanded string—no expansion is performed on the argument: `echo ~root` yields `/var/root`, while `x=root ; echo ~$x` yields `~root`. *Parameter expansion* uses a separate set of transition rules, $\mathsf{sys}, s, k \longrightarrow \mathsf{sys}', w$, meaning that under the system $\mathsf{sys}$, the variable $s$ expands in format $k$ to the new system $\mathsf{sys}'$ and word $w$ (CPARAM). Parameter expansion rules have names of the form P*. Each rule for the parameter mode has two rules: one where the variable $s$ is set and one where it isn't (e.g., PDEFAULT, PDEFAULTN, respectively). *Command substitution* has $\$(c)$ expand immediately to a special runtime form $\mathsf{cmd}\, c\ w$ (CCMDSTART). The $\mathsf{cmd}\, c\ w$ form tracks the progress of the command $c$ while collecting its output $w$ (CCMDSTEP); when the command finally terminates by reaching the no-op command :, we return the collected output (CCMDDONE). The rule sketch here omits the details of *quote expansion*, but our Lem model carefully tracks double quotes. Finally, *arithmetic expansion* runs in two steps: first the arithmetic term is fully expanded (CARITHSTEP); then the expanded term is parsed, runs (possibly side-effecting the environment), and produces a number, which is converted to a string (CARITHRUN). Since parsing happens after expanding an arithmetic term, it's possible to write new arithmetic programs at runtime, e.g., `op='+' ; echo $((1 ${op} 2))$` will output 3.

**String expansion** $\boxed{\mathsf{sys}, w \longrightarrow \mathsf{sys}', w'}$

$$\frac{e \neq k \qquad \mathsf{sys}, w \longrightarrow \mathsf{sys}', w'}{\mathsf{sys}, ew \longrightarrow \mathsf{sys}', ew'}\ \textsc{Step} \qquad \frac{\mathsf{sys}, k \longrightarrow \mathsf{sys}', w'}{\mathsf{sys}, kw \longrightarrow \mathsf{sys}', w'w}\ \textsc{Control}$$

$$\frac{\mathsf{sys}.\rho.\mathsf{env}(\texttt{HOME}) = s}{\mathsf{sys}, \sim \longrightarrow \mathsf{sys}, \text{“}s\text{”}}\ \textsc{CHome} \qquad \frac{\mathsf{sys}.\rho.\texttt{getpwnam}(s)\texttt{->pw.dir} = s}{\mathsf{sys}, \sim s \longrightarrow \mathsf{sys}, \text{“}s\text{”}}\ \textsc{CUHome}$$

$$\frac{\mathsf{sys}, s, \phi \longrightarrow w}{\mathsf{sys}, \$\{s|\phi\} \longrightarrow \mathsf{sys}, w}\ \textsc{CParam} \qquad \frac{}{\mathsf{sys}, \$(c) \longrightarrow \mathsf{sys}, \mathsf{cmd}\, c\ \epsilon}\ \textsc{CCmdStart}$$

$$\frac{\mathsf{sys}[\rho.\mathsf{fd}0 \mapsto \epsilon], c \longrightarrow \mathsf{sys}', c' \qquad \mathsf{sys}'.\rho.\mathsf{fd}0 = w'}{\mathsf{sys}, \mathsf{cmd}\, c\ w \longrightarrow \mathsf{sys}'[\rho.\mathsf{fd}0 \mapsto \mathsf{sys}.\rho.\mathsf{fd}0], \mathsf{cmd}\, c'\ ww'}\ \textsc{CCmdStep} \qquad \frac{}{\mathsf{sys}, \mathsf{cmd}:\ w \longrightarrow \mathsf{sys}, w}\ \textsc{CCmdDone}$$

$$\frac{\mathsf{sys}, w \longrightarrow \mathsf{sys}', w'}{\mathsf{sys}, \$((w)) \longrightarrow \mathsf{sys}', w'}\ \textsc{CArithStep} \qquad \frac{\mathsf{eval}(\mathsf{sys}.\rho.\mathsf{env}, \mathsf{parse}(f)) = \mathsf{env}', n}{\mathsf{sys}, \$((f)) \longrightarrow \mathsf{sys}[\rho[\mathsf{env}']], \mathsf{itoa}(n)}\ \textsc{CArithRun}$$

**Parameter expansion** $\boxed{\mathsf{sys}, s, k \longrightarrow \mathsf{sys}', w}$

$$\frac{\mathsf{sys}.\rho.\mathsf{env}(s) = s'}{\mathsf{sys}, s, \mathsf{default}\, w \longrightarrow \mathsf{sys}, s'}\ \textsc{PDefault} \qquad \frac{s \notin \mathrm{dom}(\mathsf{sys}.\rho.\mathsf{env}) \qquad \mathsf{sys}, w \longrightarrow^* \mathsf{sys}', w'}{\mathsf{sys}, s, \mathsf{default}\, w \longrightarrow \mathsf{sys}', w'}\ \textsc{PDefaultN}$$

Figure 5: Abridged semantics: expansion

**Command evaluation.** Commands have a more conventional semantics via the rules $\mathsf{sys}, c \longrightarrow \mathsf{sys}', c'$: a system configuration sys and a command $c$ take a step to a new configuration and reduced command (Figure 6). Simple commands $\overrightarrow{s = w}\ \vec{w}$ can execute one of fours ways: as a pure assignment, when $\vec{w}$ is empty (Assign); as a built-in command, like `cd` (no rules shown); as a user-defined function (no rules shown); or, as a program (no rules shown). However they're run, simple commands first expand everything from left to right; then either assignments are made globally (Assign) or just for the given command (otherwise). Structured control rules—we show them for conditionals, but would need them for sequential composition `;` and the logical operators—are fairly standard (If*). The only shell-specific feature is that the truth or falsehood of command results is determined by looking at *exit codes*, which are stored in the shell's environment in the special variable `?` (we write $\mathsf{sys}.\rho.\mathsf{env}(\texttt{?})$). Just as C treats ints as booleans, so does the shell—though the conventions are exactly opposite. In C, everything but 0 is considered "truthy", while 0 is "falsy"; in the shell, 0 is "truthy" while everything else is "falsy".

We offer two rules that are shell specific. *Subshells*, which we write $\mathsf{sub}\, c$ but have concrete syntax `( c )`, run the command $c$ in a copy of the current shell environment, $\mathsf{sys}.\rho$ (Subshell). The processes and filesystem might change, but the outer shell environment $\mathsf{sys}.\rho$ remains untouched. Following the implementation in dash [17], subshells run in a new process, with process id (pid) $p$—the main shell uses the built-in command `wait` to wait on the process $p$. *Background commands*, written $c\&$, run the command $c$ in the background. Like for a subshell, we fork a new process with pid $p$; but the outer shell returns immediately, storing the pid $p$ of the background process in the special variable `!`.

The semantics here is a sketch, omitting many rules. Adding while and for loops, user-defined functions, and control built-ins like `break` and `return` is a matter of adding some kind of explicit stack. Adding pipes and redirects requires that we model file descriptors and their corresponding

**Command evaluation** $\boxed{\mathsf{sys}, c \longrightarrow \mathsf{sys}', c'}$

$$\frac{\mathsf{sys}, w_1 \longrightarrow^* \mathsf{sys}_1, f_1 \quad \ldots \quad \mathsf{sys}_{m-1}, w_m \longrightarrow^* \mathsf{sys}_m, f_m}{\mathsf{sys}, s_1 = w_1 \ \ldots \ s_m = w_m \longrightarrow \mathsf{sys}[\rho.\mathsf{env}[s_i \mapsto \mathsf{str}(f_i)]], :} \textsc{Assign}$$

$$\frac{}{\mathsf{sys}, \mathsf{sub}\, c \longrightarrow \mathsf{sys}[P \uplus (p, c, \rho)], \mathtt{wait}\ p} \textsc{Subshell} \qquad \frac{\mathsf{sys}, c_1 \longrightarrow \mathsf{sys}', c_1'}{\mathsf{sys}, \mathsf{if}\ c_1\ c_2\ c_3 \longrightarrow \mathsf{sys}', \mathsf{if}\ c_1'\ c_2\ c_3} \textsc{IfStep}$$

$$\frac{\mathsf{sys}.\rho.\mathsf{env}(?) = 0}{\mathsf{sys}, \mathsf{if} : c_2\ c_3 \longrightarrow \mathsf{sys}, c_2} \textsc{IfTrue} \qquad \frac{\mathsf{sys}.\rho.\mathsf{env}(?) \neq 0}{\mathsf{sys}, \mathsf{if} : c_2\ c_3 \longrightarrow \mathsf{sys}, c_3} \textsc{IfFalse}$$

$$\frac{}{\mathsf{sys}, c\ \& \longrightarrow \mathsf{sys}[P \uplus (p, c, \rho)][\rho.\mathsf{env}[! \mapsto p, ? \mapsto 0]], :} \textsc{Background}$$

Figure 6: Abridged semantics: commands

buffers. The rules for command substitutions (CCmd* in Figure 5) make a gesture in this direction. I've chosen Lem in particular because the SibylFS formalization of POSIX file systems is in Lem [62]—I plan to use them to model our filesystems FS and file descriptor handling.

### 3.1.3 Getting it right

At the time of writing, we have a more or less complete Lem model of expansion itself, but have not yet addressed command evaluation. There are three ways we can check our model of the POSIX shell: the POSIX test suite, comparing to existing shells, and proving theorems.

The POSIX test suite is a series of testing scripts that run in a shell to determine the degree to which it complies with the POSIX standard [40]. The shell testing scripts consist of 23k SLOC of scripts, with other tests for testing built-ins and core utilities. Since the tests themselves are written in shell as `expect` scripts, truly executing these tests in our model would be challenging—we would need to be able simulate running external programs! We could, however, manually wrap an extraction of our model to be a functional shell and test that; alternatively, we could try to extract tests in a format more amenable to our model. We've written hundreds of our own tests but haven't yet extracted ground-truth test cases from the POSIX test suite.

While the POSIX test suite is authoritative in terms of the POSIX standard, programmers are more familiar with particular, concrete shells, like dash and bash. We can follow the lead of the methodology for testing the $\lambda_{\mathsf{JS}}$ compiler, which compiles JavaScript into a more conventional language [35]: we can compare our model's behavior to existing shells on a test suite. Jeannerod et al. have been collecting a large (30k program) test suite, as well [42].

The first thing I plan to do with the semantics is to create a compiler in the $\lambda_{\mathsf{JS}}$ style, translating shell scripts to a more conventional first-order language with explicit calls to expansion routines (Section 3.2). This compiler could be tested in the POSIX test suite or against existing shells—while such testing would only indirectly tell us something about our formalization, successful tests would increase confidence in our semantic foundations. We might be able to emulate SibylFS's nondeterministic approach: the SibylFS formal model of the POSIX filesystem predicts a range of allowable behaviors for each filesystem call, and they test their model by comparing real-world behaviors to those allowed by the model [62].

Finally, we can check that our semantics are coherent by proving theorems. Lem's extraction

to Coq makes our formal model amenable to foundational proof. I can imagine proving simple properties—e.g., the only side effect of expansion without command substitution is setting variables and errors—or, more ambitiously, verifying that our shell compiler is semantics preserving.

### 3.2 Tools for the POSIX shell

The formal semantics in Lem can be extracted to executable OCaml or Coq definitions, making the semantics themselves a plausible basis for building tools and proving theorems. There are two tools in particular that seem within the scope of the grant—a compiler and a static analysis—but I can imagine several other ways to use our semantics to develop tool support for the POSIX shell.

First, I plan to use the semantics to write a compiler in the style of $\lambda_{JS}$ [35]: compile shell scripts to a more sensible, first-order intermediate language, where all of the expansion and file descriptor management is done via explicit calls to library functions. Translating to a conventional intermediate language not only helps us think clearly as programmers, but making the shell's behavior explicit also reduces the friction for using conventional tools on translated shell programs. Our compiler will also offer the usual benefit: a speedup over interpreted code. While a backend in C is a natural choice, a backend in Python or OCaml that produced clean code would help migrate from scripts to programs; the Turtle library for Haskell would be a natural backend [28].

Second, the semantics will guide the implementation of a static analysis. My goal is to be able to analyze a shell script for common bugs—improperly handling filenames with whitespace, for example—as well as deeper checks, e.g., determining under what circumstances the script will call `rm` and with which arguments. Regular expressions make a natural abstract domain: each primitive command has a "type" specifying which invocations are acceptable, accepting some language of strings as input and producing some possible string of system calls and outputs. While regular languages can't hope to capture all of the complexity, the simplest possible checks would have detected the Steam cleaning bug that wiped hard drives [58]. As a goal for evaluating how good the analysis is, I hope to be able to summarize output a `curl`-based installer, saying: this script will download a tarball from a given URL, untar it, and move files into `/opt` and `/var`. Summaries could also include information flows, such as, "this script reads from `/etc` and then fetches a URL based on information read from there". To be clear, it is probably too hard to write a static analysis that will defeat a determined attacker: we can defend against carelessness, not maliciousness.

I propose to construct both the compiler and the static analysis. These are not small projects, but both tools will be a platform for research. Once we have the formal semantics and tools for working with it, we can apply ideas from elsewhere in PL. For example, I've already started constructing an interactive tool for learning about shell expansion (Section 4). An automatic tool could detect brittleness in shell scripts and synthesize the "hardening" necessary—signal handling and error handling, avoiding clobbering files, etc. Such a tool could even be integrated into the compiler. The static analysis could be used to support the script development process: the regular expression domain matches the existing completion infrastructure common in modern shells [4, 21]. The analysis could also support script development. Shell scripts are often developed step-at-a-time, printing out intermediate values before running commands (Section 2.3); the analysis would allow us to print out not just specific concrete values, but symbolic ones characterizing how the program might run in a different environment.

### 3.3 Designing interactive programming languages

The concrete, immediate goals of the proposed work are to better understand and support the shell—but this concrete work will also pay abstract dividends: how should we design an interactive

language? Which language features promote interactivity? Are there specific constellations of language features that make something more interactive—or more error prone? The last part of the research will be to study interactive language design by way of the shell, gleaning ideas applicable in other "live programming" settings.

There are three levels at which we can study interactive language design: from a generalist perspective, focusing on perennial questions of scope and top-level definitions; from a low-level perspective, focusing on small, mostly non-breaking repairs to the POSIX shell; and from a mid-level perspective, focusing on reimaginings of the shell's semantics that break significantly from the POSIX standard.

**Interactive programming.** Designing the interactive parts of languages is hard: "the top-level [interactive REPL] is hopeless", as it leads to what amounts to dynamic scope [23, 72]. But the top-level is also an essential tool for exploration, development, testing, and debugging. Interactive programming is even more important in other settings: scientists use Matlab and R's interactive modes to explore data sets and develop programs; musicians use programs to control audiovisual performances as they happen. What is the right model of interactive programming? While dynamic scope is, in general, not a good idea, is there some way to tame the dynamicity? What insights can get from the shell? Can we apply them to other settings, like scientific programming workbenches and music? Can we apply them to general purpose languages?

**Cleaning up the POSIX shell.** The details of the shell have been meticulously kept up-to-date with changes in POSIX, but the language is, at its core, nearly 30 years old—going back to the Bourne shell, which is itself 40 years old. The programming languages community has learned a great deal about language design in the past 40 years, and the shell has not benefited from it. Others have reevaluated POSIX's abstractions [2]; can we come up with a variant semantics that *mostly* conforms to the POSIX standard but avoids some of the more grotesque features? We offer two examples of POSIX shell "warts" that could, we conjecture, be safely fixed: `alias` and insufficient quoting modes. Perhaps we can avoid the "semantic bloat", treating rarely used features unsoundly—without any repercussions [48, 49].

As a first example, the `alias` facility allows shell users to change the way lexing works [42]. For example, writing `alias nope=for` will cause the token `nope` to appear to the lexer as the reserved keyword `for`, so `nope x in 1 2 3; do echo $x; done` will produce three lines, reading 1, 2, and 3. We can alias a single token to multiple tokens, as in `alias why=for x in`, and then `why 1 2 3; do echo $x; done` will produce the same output, with `x` unhygienically captured. Finally, `alias` has side-effects for parsing future commands, not the current command: running the single, compound command `alias why="for x in"; why 1 2 3; do echo $x; done` will cause a parse error, but running the alias and the loop separately succeeds. If a shell script uses `alias`, it's effectively impossible to statically *parse* the script, let alone assign the script any meaning! Fortunately, `alias` seems to mostly be used to save keystrokes in the interactive mode or to force certain flags by default (e.g., defaulting the `rm` command to `rm -i`, so that the user is prompted). A cursory search of GitHub found that `alias` isn't used for the dangerous token shenanigans in the example above—it's mostly used in ways that could be replaced by user-defined functions.

Second, the shell's treatment of field separators make it difficult to handle filenames with spaces in them. Consider the following interaction, where the user is trying to delete the file called `filename with spaces` in a directory where there are two other files, `filename` and `spaces`:

```
$ ls -1 # -1 means "one file per line"
filename
```

```
spaces
filename with spaces
$ x="filename with spaces" # can we make x a single field at defn?
$ rm $x # no! just deleted "filename" and "spaces"
rm: with: No such file or directory
$ ls -1
filename with spaces
$ rm "$x" # need quotes at use site
```

It's not possible to specify at `x`'s definition site that it holds a single field. Field splitting is entirely controlled by use sites—changing the use site `rm $x` to `rm "$x"` gets the right behavior. Worse still, sometimes we can't control expansion at all, as in the quite common loop `for x in ‘ls‘; do some_command $x; done`. Only by obscure manipulations of the `IFS` variable (or by using `find -print0`, or some other arcane incantation) can we properly process each file in a directory when those filenames have spaces in them. Can we subtly change the field-splitting semantics in ways that most scripts won't notice, making it possible to handle whitespace in filenames gracefully?

The research will surely produce other examples of ill-designed features in the POSIX shell. We can get a sense of how frequently these features are used by looking at publicly available code on GitHub, which will offer guidance in terms of what can be easily repaired and what requires that we start from scratch.

**A new shell.** It may be that the only way to a sensible semantics for a shell is break significantly with POSIX. I am particularly interested in designs for alternative shells that keep *both* interactivity and scriptability. Non-POSIX shells are popular (csh/tcsh, fish), but barring the use of tcsh in BSDs, these shells are mostly just for interactive use. Academic efforts, like scsh and Shill [56, 69], have focused on producing languages that can perform shell-script-esque tasks, but have lost the interactive feel.

While I doubt that a wholesale reinvention of the shell is the right choice, radical new designs shouldn't be entirely ruled out. For example, the shell's field splitting forgets about field splits at every expansion. That is, shell variables are always of type string... can we allow for more structured values, with automatic coercions to strings? While automatic conversions between types are always fraught, such a design might allow an improved shell to retain a fair proportion of backwards compatibility while offering revised, cleaner semantics.

## 4 Education plan

The broadest educational impact of this work will be the training materials—tutorials, examples in the spirit of The Command Line Murders [73], and ultimately short videos—I plan to develop, ranging from introductory material for undergraduates and new system administrators up to best practices for experienced shell programmers. I will develop these materials jointly with undergraduates and the Pomona College IT department, who are beginning to train its Windows administrators in Linux, producing the videos near the end of the grant; the open source materials will be hosted and distributed by the Liberal Arts Consortium for Online Learning (LACOL) and made freely available.

Beyond the teaching materials and videos, I will use insights from the research to help revise the Pomona College curriculum. At present, the Pomona College requires students to use the shell, mostly in the second and third courses and in more advanced systems courses. But there's

little curricular organization around how one uses the shell, how one writes scripts or Makefiles, or what the pitfalls are; we're so careful and principled around teaching programming, but are frankly sloppy when it comes to the shell. As part of the proposed work, I will work with my colleagues to unify our CS curriculum's treatment of the shell, giving it the same level of care we give to other CS activities. I have already run two workshops on the shell—one as a broad introduction and one as an introduction to shell scripting—which were well-enough received that they were mentioned by students as "the sort of thing we need more of" during a recent departmental review. Similar workshops will test materials on Pomona College IT staff and Claremont College researchers interested in data science the Digital Humanities; feedback from these workshops will guide the creation of the videos.

# 5 Broader impacts

The proposed research holds promise for several communities: system administrators and programmers, researchers, and with educational impact on the students, faculty, and staff at Pomona College and elsewhere. Tools for eliminating shell bugs will help system administrators and programmers; more powerful semantic analysis tools for shell would be of immense interest in practice. The formal semantics will serve as a platform for research: the possible support tools one can build are too numerous for me to build them by myself, but the Lem model will allow others to use my semantics, too. Initial research efforts have already involved two undergraduates at Pomona College; the educational material produced will help students at Pomona College and beyond, with the videos extending the reach of the research into industry. The students involved in the research and coursework will come from underrepresented minorities, as well as students from low-income families, and first-generation college students: Pomona College is dedicated to "creating a dynamically diverse community" [61] and admits and supports students from less privileged backgrounds. Miniaturized versions of shell features (expansion, pipes, redirection) will be adapted for coursework. All of the research products produced will be open source and freely available.

# 6 Related work

**Foundational semantics.** Mazurak and Zdancewic gave a static analysis for some of bash's expansion's with an eye towards security [52]. Our model of expansion already goes further than theirs—we model arithmetic expansion. Jeannerod et al. have begun building a model of the shell under a slightly different approach [41]: rather than building a conventional core language and making the shell explicit, they've crafted a core language with a shell-like semantics, building a verified interpreter in Why3. In their efforts to verify shell programs, they have also begun parsing large sets of shell files [42]. Our approaches are in principle similar, though our methods and end goals are different. I propose to build a full semantics for the shell, not for a core, shell-like language—I believe such a semantics will enable research just as foundational semantics for *full* JavaScript [35, 9] and C [7, 46] have. In order to be truly confident that the core is correct, we need a semantics for the high-level language, and their research omits that crucial step. Finally, while we're both interested in static analysis for installation scripts, I am also interested in the interactive shell. Our research projects will be complementary.

The subshell construct $(c)$ (in our abstract syntax, $\mathsf{sub}\, c$) amounts to a limited form of Warth et al.'s "worlds"—first-class, programmatic heaps [74].

We can build on any one of a variety of formalizations of the POSIX filesystem [22, 26, 59, 62]; other systems can in turn use our semantics to reason about shell programs [68]. Ridge et al.'s

model [62] makes the Lem tool [57] more immediately appealing than K [64].

**Tool support.** A variety of tools help programmers navigate the POSIX shell's hazards [45, 44, 76]. Unfortunately, existing static analyses work only at a shallow, syntactic level; existing dynamic analyses only offer limited guarantees. D'Antoni et al. use machine learning to repair simple commands, but they offer no semantic insights [16]; can we improve on their synthesis/repair-by-example approach by knowing more about the shell's semantics?

Others in academia and industry have invented new shells, but they are either very far from the interactive POSIX shell (EZ [24], Levine's LISP shell proposal [47], REXX [15], scsh [69], shill [56]) or just a veneer over it (fish [21]; PWB [51]).

Finally, there is a robust literature on static analysis for the way strings are used in scripting languages like PHP [55, 43, 77, 75, 79, 19, 78, 80, 71] but also more conventional languages like C [25] and Java [81]. We will reuse the core of these analyses by compiling the shell into forms they can accept.

**Interactivity.** Relevant studies of interactivity can be broken into a few groups: exploratory or experimental programming [65]; interactive programming [66, 70, 53, 54]; and live coding for musical performance [14][1] or design [10, 11, 37, 12]. Recent PL work on prodirect manipulation and synthesis-by-example offer promising strategies for helping shell users automate their tasks [36, 12].

# 7 Results from prior NSF support and PI qualifications

I have not been a PI on any NSF awards; I was the beneficiary of a Schloss Dagstuhl–NSF Support Grant for Junior Researchers to attend Dagstuhl Seminar 16131 in March 2016. Throughout my career, my work has used foundational semantics to understand software systems. My PhD was largely supported by NSF award 0915671, under which we published several foundational papers in contract semantics while also building contracts into Boomerang, a bidirectional programming language (itself supported by NSF award 0534592), as well as Breeze, a dynamically information flow language [33, 3, 34, 6, 38]. Later work on software-defined networking was supported by NSF awards 1016937 and 1111520: we built foundational semantics for network policy languages and implemented them as a variety of compilers [67, 5]. Several of my papers use Coq [18] or QuickCheck [13] to mechanically verify or test formalizations [33, 34, 38, 29], and I am also a co-author of a textbook on using Coq to study logic and programming language theory [60]. In my brief time at Pomona College, I have worked with eight undergraduates—half of whom are underrepresented minorities in computer science—and have had success encouraging them to apply to and attend graduate school. Finally, it is commonplace to mock the shell as "irredeemably bad", but I do not share this opinion. Beyond my technical qualifications, I am a frequent and enthusiastic shell programmer—I hope to somehow help redeem the POSIX shell.

# 8 Summary

The POSIX shell is neglected as an object of study, even though it is a source of costly mistakes. Bringing programming-languages research techniques to bear on the shell will allow us to support and improve the shell—and along the way, we will learn more about interactive programming and language design.

[1] The Live Code Research Network is a good starting point for learning about this community.