Word expansion supports POSIX shell interactivity

Submission for publication

Michael Greenberg
Pomona College
Claremont, CA, USA
michael@cs.pomona.edu

ABSTRACT

The POSIX shell is the standard tool to deploy, control, and maintain systems of all kinds; the shell is used on a sliding scale from one-off commands in an interactive mode all the way to complex scripts managing, e.g., system boot sequences. For all of its utility, the POSIX shell is feared and maligned as a programming language: the shell is feared because of its incredible power, where a single command can destroy not just local but also remote systems; the shell is maligned because its semantics is non-standard, using *word expansion* where other languages would use *evaluation*.

I conjecture that word expansion is in fact an essential piece of the POSIX shell's interactivity; word expansion is well adapted to the shell's use cases and contributes critically to the shell's interactive feel.

CCS CONCEPTS

• **Software and its engineering** → **Scripting languages**; **Command and control languages**; *Language features*; *Semantics*; • **General and reference** → *Design*; • **Human-centered computing** → *Command line interfaces*;

KEYWORDS

command line interface, interactive programming, word expansion, string manipulation, splicing, evaluation

ACM Reference Format:
Michael Greenberg. 2018. Word expansion supports POSIX shell interactivity: Submission for publication. In *Proceedings of Programming Experience Workshop (PX/18)*. ACM, New York, NY, USA, Article 4, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

PX/18, April 2018, Nice, France

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Command-line interfaces are the expert's way of exercising control over their computer: installing, configuring, and removing software; creating, moving, deleting, or otherwise manipulating the filesystem; deploying, monitoring, and shutting down services. The foregoing tasks are often easier done in a shell; sometimes, these tasks *must* be done in the shell, for lack of other options.

While other shells exist, the POSIX shell is the *de facto* standard [9]; I'll simply refer to it as "the shell". As a programming language, the POSIX shell has several distinctive features [3]: it excels at controlling concurrent processes; it is used along a continuum from interactive command-at-a-time use to batching of commands to lightweight scripting all the way to programming of system-critical scripts; it is programmed in an exploratory, "print what you do before you do it" fashion; shell scripts have the computer literally do what a human would; and, its semantics mixes conventional evaluation with *word expansion*. I am particularly interested in understanding this last feature: what is word expansion, and how is it essential to the POSIX shell?

In this paper, I explain what word expansion is (Section 2) and offer arguments for it being a quintessential interactive shell feature.

I offer two positive arguments (Section 3): first, the shell's core abstractions for managing processes are string-based, and word expansion has convenient defaults for combining strings (Section 3.1); second, the commands run in the shell have calling conventions that encourage the use of, if not word expansion itself, an expansion-like mechanism (Section 3.2). I also offer negative arguments (Section 4): two academic shell 'replacements' (scsh and Shill, neither of which use word expansion [14, 15]) have shown their merit as replacements for the shell as a programming language, but not as interactive tools. Similarly, the fish shell replacement works well as an interactive shell but less popular for programming [7]. Shell-like libraries seem to a do a good job for scripting, but less so for interactive fork (Section 4.2): a shell library for Python, Plumbum, ends up falling back on word expansion [8]; a shell library for Haskell, Turtle, doesn't quite work as an interactive shell [2].

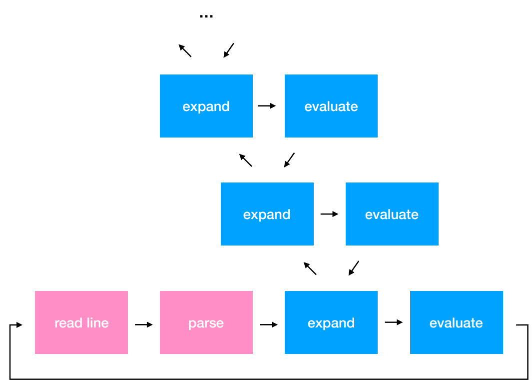

Figure 1: The shell REPL, with parsing in pink and execution in blue. Execution consists of expansion followed by evaluation, but expansion can embed further executions using *command substitutions.*

My arguments are by no means exhaustive: we might assess how important word expansion is in other ways (Section 5), and we might make word expansion better or less error prone without fundamentally changing its character (Section 6).

The technical parts of the paper are, for the most part, a recapitulation of the POSIX standard [9]. My arguments reflect my own bias towards a *semantic* understanding of the shell. I use my own experience as evidence; however, there are other good forms of evidence: historical analysis of various other shells, user studies, and experiments in shell design, to name a few.

2 WHAT IS WORD EXPANSION?

The POSIX shell executes somewhat unconventionally (Figure 1). Like other dynamically typed, interactive languages, the shell operates in a "read-eval-print loop", or REPL. But the shell's evaluation is split into two phases: first, a phase of *word expansion*, followed by a second phase of actually evaluating code. What's more, word expansion can itself trigger expansion and evaluation recursively. Those who are *very* familiar with the shell may well skip the next section and go directly to Section 3. Those who use the shell in a less thoroughgoing way may benefit from the following high level overview of its (commonly misunderstood) features.

Word expansion is specified in Section 2.6 of the POSIX IEEE Std 1003.1-2008 [9]. At a high level, word expansion is the process that converts user input into *fields*, which will become, e.g., a command and its arguments. There are seven stages of word expansion:

(1) *tilde expansion*, which replaces `~` with the current user's home directory and `~user` with a given user's home directory;
(2) *parameter expansion*, which replaces variable references like `$x` with the value of the given variable, possibly applying one of a number of *formats*, e.g., `${x=5}` will either return the value of `x` or, if `x` is unset, it will assign the result of recursively expanding 5 to `x`;
(3) *command substitution*, which nests evaluation inside of expansion by running a given command, e.g. `` `cmd` `` or `$(cmd)` will splicing in `cmd`'s output via the recursive expansions and evaluations in Figure 1;
(4) *arithmetic expansion*, which computes the result of an arithmetic expression, e.g., `$((x += 2))` will add 2 to the current value of `x` (interpreted as a number) and return the string representing the number two greater than `x`;
(5) *field splitting*, which breaks the expanded input string into fields;
(6) *pathname expansion*, which uses the current working directory to expand special symbols like `*` and `?`; and
(7) *quote removal*, which removes any double quotes that the user used to control field splitting.

The first four stages are properly expansions on user input and are run in a left-to-right fashion; the last three stages arrange for splitting the string into fields. It seems typical of shell implementations to perform all seven stages in one go from left to right, generating a linked list of fields.

For example, suppose we were to run the following command:

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

There are three *control codes* subject to expansion:

- `${x=5}` will expand via parameter expansion; if `x` is set, then it will return the current value of `x`; if not, the string `5` will be expanded (to itself), set as the value of `x`, and then it will return the new value of `x`, viz., 5.
- `$((x+=1))` will expand via arithmetic expansion, adding 1 to `x`'s value.
- `${x}` will expand to `x`'s current value (or the empty string, if `x` is unset.

In this example, expansion runs as follows if `x` is unset:

```
echo ${x=5} $((x+=1)) ${x}
echo 5      $((x+=1)) ${x} # x set to 5
echo 5      6         ${x} # x set to 6
echo 5      6         6
```

Field splitting will generate four fields: one for `echo`, one for `5`, one for the `6` that came out of arithmetic expansion, and one for the `6` that came out of the final parameter expansion.

Word expansion is subtle in terms of (a) the order of events, and (b) the nature of field splitting.

For an example of the subtlety of the order of events, consider the string `$((1 $op 2))`. Before arithmetic expansion can begin, the string `1 $op 2` must be fully expanded so it can be parsed as an arithmetic expression. If `op` is bound to a valid binary operator, like `+`, then the fully expanded string `1 + 2` will parse and evaluate to `3`. If, however, the variable `op` is unset, then `$op` will expand to the empty string, and the string `1 2` will fail to parse. (We'd find a similar failure of `op` produced something other than operator, like `hello` or `47`.) The issue isn't only with arithmetic substitution: other forms of expansion have nested expansion in them. Using command substitution, a word expansion can trigger multiple layers of expansion and evaluation, e.g., `$(echo ${x=$(echo 5)})` will begin by trying to expand `${x=$(echo 5)}`; if the variable `x` is unset, it will then run a nested command substitution on `echo 5`, after which it will update the value of `x` and run the outer command substitution—the recursive expansion/evaluation shown in Figure 1.

For an example of field splitting being subtle, suppose `x` is bound to the string `a␣b␣c` (where ␣ represents a space). By default, `${x}` would expand to three fields: one for `a`, one for `b`, and one for `c`. If the user sets the `IFS` variable, the *internal field separators* can be configured so that `${x}` would expand as a single field, retaining spaces. Understanding which and how many fields will be expanded can be challenging, and the defaults are particularly awkward for filenames with spaces. For example, suppose we have a directory with three files: one called `file1`, one called `file2`, and one, unfortunately, called `file1␣file2`. If we set `x` to `"file1␣file2"` and run `rm ${x}`, we might be in for a surprise: `x` expands to two fields and the first two files are deleted! Putting the variable substitution in quotes solves the problem: `rm "${x}"` will delete only `"file1␣file2"`. That is, field splitting can be controlled at use sites but not at definition sites.

2.1 Word expansion in evaluation

Expansion aside, the shell's *evaluation* model is fairly conventional for its control operators: sequence (`...; ...`), conditionals (`if ...; then ...; else ...; fi`) and while loops (`while ...; do ...; done`) work as expected. The shell also supports some operations for controlling processes, like short-circuiting conjunction (`... && ...`) and disjunction (`... || ...`). Along with negation (`! ...`), these logical operators use commands' exit codes to determine conditionals, noting that the notion that a command is 'truthy' when it yields an exit code of 0. Pipes set up file descriptors from one process to another (`... | ...`). None of these command forms make particular use of word expansion in their semantics.

Four shell forms deal concretely with word expansion in their semantics: redirections, simple commands, for loops, and case statements.

Redirections set up file descriptors for a single process (`... >...`, etc.). The targets of redirections are generated by word expansion. For example, `echo hi >$f` will:

(1) run word expansion on `$f` to find out which file should be used—here, whatever the variable `f` holds, collapsing the list of expanded fields to a string;
(2) create a new process with the standard out file descriptor (file descriptor number 1) redirected to the resulting word expansion; and
(3) run the echo command (which could either be an executable on the system, e.g., `/bin/echo`, or a built-in command in the shell).

Simple commands depend even more heavily on word expansion. Simple commands have the shape of zero or more assignments followed by zero or more arguments: `VAR1=val1 VAR2=val2 ... VARm=valm arg1 arg2 ... argn`. Each `val` and `arg` is subject to expansion, which is performed from left to right. (The variable names `VAR` are statically known strings and neither an input nor an output of expansion.) If there are no `args`, then the variables are assigned globally in the shell environment. If there are any `args`, then the variable assignments have a more restricted scope, and the shell evaluates as follows:

(1) Every `val` is expanded, but the environment isn't updated yet.
(2) Every `arg` is expanded. The very first field is used to determine which command is being run, where each command could be either (a) an executable somewhere on the system, (b) a function call, or (c) a shell built-in.
(3) In the case of (a) and (b), each `VARi` is bound for the result of expanding `vali` when running the command or calling the function. In the case of (c), shell built-ins do not typically look at the environment, but some special built-ins will update the environment with the variable bindings (Section 2.14 [9]).

For loops and case statements use word expansion to determine control flow. The loop `for x in args; do ...; done` begins by expanding `args`; after splitting the expanded `args` into some number of fields, the body of the loop is run with `x` bound to each resulting field in turn.

Case statements `case args in pat1) ... ;; pat2) ...;; esac` evaluate by expanding `args`, collapsing the split fields into a single string, and attempting to match the resulting string against each pattern, `pat`, in the given order. When a pattern *matches* against the string, the commands in that branch are run and the other branches are ignored. In this context, matching is a limited form of regular expressions, where the star pattern `*` matches an arbitrary span of characters and ? matches any single character. The shell also permits alternation in patterns, as well as various locale-defined character classes.

Only four command forms make particular use of word expansion, but it still turns out that executing nearly any command will require some number of word expansions: simple commands are in some sense the “base case” of the recursive evaluation function. Up to a first approximation, though, it’s more or less sound to imagine the shell has a standard evaluation semantics. When field splitting is involved, however, the shell lives up to its reputation for unpredictability.

In the remainder of this paper, I argue that word expansion as a critical enabling feature for the POSIX shell. The shell is successful both as an interactive way of controlling a computer—and word expansion supports that interactivity.

3 WHY IS EXPANSION IMPORTANT?

Word expansion is a critical, enabling component of the POSIX shell: the shell’s niche is fundamentally about string processing, and word expansion is a good default for the operations the shell invokes.

3.1 The shell’s core abstractions

The POSIX shell is fundamentally about managing processes and their file descriptors: commands create processes; redirections and pipes arrange file descriptors; the various control primitives like `for`, `do`, and user-defined functions serve to automate process management. The core process management tasks, however, are all about strings: the strings used to specify a command and its arguments to `execve`[1], the strings used to refer to filesystem locations, the strings that are the contents of important files in UNIX, and the strings that are the values of environment variables.

While the ultimate goal of the interactive shell is job control—starting and stopping programs—the job control process is itself all about strings. Languages like Perl, Python, and JavaScript all have good support for string manipulation in the language and standard library; these languages include some string manipulation features that the shell lacks, and all three make do without word expansion. Nevertheless, all three are unsuitable for interactive use as a shell and are less suited for job control (but see Section 4.2).

3.2 The shell’s operators and operands

Two characteristics of the shell make word expansion particularly useful: first, more things are operators than operands in the shell; second, the POSIX shell’s operators tend to be *variadic*—commands like accept anywhere from zero or more (`ls`), one or more (`rm`), up to two or more (`mv` and `cp`) arguments. These variadic commands are particularly well suited to word expansion, which produces multiple arguments.

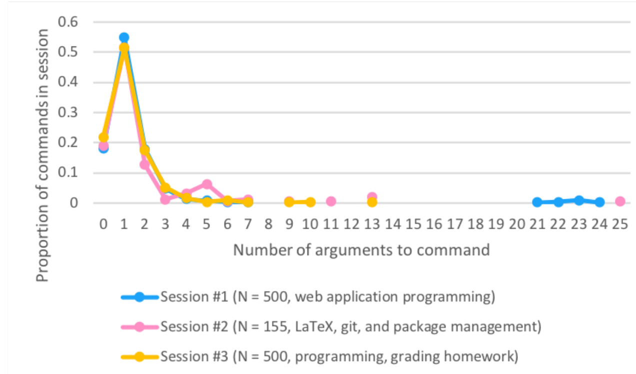

Figure 2: Three sessions of interactive work in the shell; more than 75% of all commands take at least one argument.

How might I substantiate the claim that interactive use of the shell tends to have multi-argument, variadic commands? There hasn’t been much research on how the shell is used today. So far as I can tell, all of the work examining the POSIX shell as a user interface comes from nearly thirty years ago [4, 12, 18]. Both Kraut et al.’s early analysis of UNIX shell usage and Hanson et al.’s later extension of that analysis provide valuable insight into the design of commands, though they seem to take a menu based system as a foregone conclusion [4, 12]. Their studies are more than thirty years old, track processes rather than actual shell commands, don’t account for the POSIX shell as a language (pipes `|` are treated as commands rather than command-formers), and may not reflect current usage. Wixon and Bramhall [18] offer comparative counts of commands in UNIX and VMS, but don’t keep track of how many arguments these commands were given, whether or not word expansion was triggered, etc. Their numbers are more than thirty years old, and may reflect differences in interaction styles. For example, nearly 15% of VMS commands were to open an editor, when I almost never explicitly run such a command—instead I run `open`, which calls the default OS handler for that item’s file type, or I directly open it from a separate text editor window.

Absent other sources of information, I offer a brief analysis of my own shell history. I analyzed three sessions of interactive work, finding that the vast majority of shell commands take multiple arguments and variadic commands are common (Figure 2). In the first session—programming a web

[1] The `execve` system call is how a command is run in the shell: given the path to an executable, a list of arguments, and an environment, `execve(cmd,args,env)` replaces the current executing process with the command `cmd` on arguments `args` in environment `env`.

application written in Ruby/Sinatra—an overwhelming majority of commands take at least one argument (81.9%), with more than a quarter of commands taking more than one argument (27.1%). Out of 500 commands, 38 made use of a variadic interface (7.6%). In the second session—writing LaTeX, version control with git, and package manager configuration—80.9% of commands take at least one argument, with 29.3% of commands taking more than one. Out of 155 commands, 15 made use of a variadic interface (9.7%). In the third session—programming in Haskell and C, version control with git and subversion, some package and build management, and homework grading—78.1% of commands take at least one argument, with 26.7% of commands taking more than one. Out of 500 commands, 62 made use of a variadic interface (12.4%). Summarizing these results, more things in the shell are operands than operators, and many operators take multiple arguments.

Only my third sample session contained loops—several revisions of a `for` loop for sending out emails about homework grades; I found no other programmatic constructs, like `if` or `while`. In their sample of more than 30K Debian package installation scripts, Jeannerod et al. find plenty of loops. In their setting, 59% of these for loops are directly unrollable, i.e., iterated over constant argument—that is, their loops were over constant arguments and needn't have made use of expansion at all (my loops depended on the filesystem were not unrollable) [11]. I attribute this difference to the samples: mine are drawn from interactive use, while theirs are from stylistically constrained, programmatic maintainer scripts for managing package installation.

Four of the most common commands used in a variadic way are `mv` (to move files), `cp` (to copy files), `rm` (to remove files), and `grep` (to search files). Each of these commands is *variadic*: one may supply as many arguments as one likes. My first session had 65 uses of these commands (13.0%), my second had 15 (9.75%), and my third also had 65 (15.0%). Note that these counts are slightly different from above: here I cound *every* use of these common variadic functions, whether it uses many arguments or not; above I count only those uses of *any* command with a variadic interface.

Variadic functions are far from the norm in most languages. Comparable file manipulation functions take one (`rm`, `grep`) or two arguments (`mv`, `cp`). But with interactivity in mind, variadic commands for file manipulation are ergonomic: it is quite common to treat bundles of files together. Word expansion dovetails with variadic commands: field splitting allows one to store many filenames in one variable, or to use pathname expansion to produce multiple files matching a pattern, as in `*.hs` referring to all Haskell source files.

There is a critical weakness, however, in the way the shell splits strings: the defaults use whitespace to split fields, so filenames with strings in them will be grossly mistreated. See Section 2 for an example and Section 6 for further discussion.

3.3 Interactive, exploratory programming

I frequently use the shell to automate repetitive tasks: running homework graders on students' assignments, generating grade emails, etc. Writing such scripts is fairly different from programming in conventional languages, where I tend to write large chunks of a program at a time along with its tests, checking on functionality in large batches. In the shell, I always hesitate to actually run the commands that manipulate the filesystem, for fear that something could go awry. Instead, I tend to write a script that prints out which commands *would* be run, and I can verify that those are the very commands I want to execute.

One of the main reasons for the shell's "print first, run later" paradigm is the general lack of data structures. I'm not at all afraid to add an item to, say, a list or map in my program, because data structures are ephemeral. If my program goes wrong and the data structure becomes corrupted, not much is lost—I can simply start over. But there are really only two data structures in the shell: strings (concomitant with word expansion) and the filesystem. I am *very* wary of updating my filesystem, since it's easy for a single shell command to have widespread and irrevocable effect.

Having strings as the primary data structure more or less *forces* an exploratory or interactive approach to programming. The shell's interactivity comes, in part, perhaps, from wariness of the shell itself.

4 MAKING DO WITHOUT WORD EXPANSION

I've argued that word expansion is essential to the shell's core abstractions (Section 3.1) and the shell's operators and operands (Section 3.2). We can see that word expansion is critical to interactive shell use by looking at attempts to replace the shell, in particular the academic efforts scsh [15] and Shill [14] and the popular open source shell fish [7].

Both scsh and Shill aim to replace the *scripting* portion of the shell. Shill explicitly renounces any claim to interactivity:

> Shill is not an interactive shell, but rather a language that presents operating system abstractions to the programmer and is used primarily to launch programs.

Scsh offers a similar caveat:

> It is important to note what scsh is not, as well as what it is. Scsh, in the current release, is primarily designed for the writing of shell scripts–programming. It is not a very comfortable system for interactive command use: the current

> release lacks job control, command-line editing, a terse, convenient command syntax, and it does not read in an initialisation file analogous to `.login` or `.profile`. We hope to address all of these issues in future releases; we even have designs for several of these features; but the system as-released does not currently provide these features.

Shill's focus is very much on its capability system. Shill of course supports calling arbitrary executables:

```
exec(jpeginfo, ["jpeginfo","-i",file],
  stdout = out, extras = [libc,libjpeg])
```

The first argument to `exec` is a reference to the executable to be run, which is also a capability to actually execute it (here, `jpeginfo`—we are not shown how this capability is obtained); this capability is *not* a string. Next comes the actual command as a string, and then comes the redirection (here, piping the command's output to a stream named `out`). Finally, the `extras` indicate other capabilities that will be necessary to safely run the program (here, the C standard library and a JPEG manipulation library used by the executable). Shill is very good at its job—managing capabilities—but is unsuited to interactive use. Features like the collections of capabilities they call 'wallets' ease the programmatic burden, but Shill is meant only to replace "the scripting portion of Bash".

While Shill doesn't go so far to identify precisely what makes it less suited for interactive use, scsh offers a list of features that they conjecture would foster interactive use. The list of features doesn't mention word expansion, yet I believe that word expansion is in fact critical for the interactive feel. To see why, let us consider a few common uses of expansion and compare scsh with the POSIX shell.

As a first example, consider the scsh re-implementation of the `echo` command:

```
(define (ekko args)
 (for-each
   (lambda (arg)
    (display arg) (display " "))
   args)
```

In a shell, a similar command can take advantage of the variadic echo built-in, to write:

```
ekko() { echo "$@"; }
```

To avoid tautology, we could have instead used `printf`, but in either definition, variadic commands and expansion give a simpler model than manual, programmatic iteration.

The examples get more extreme when running more complex commands. The following scsh snippet will move all of the files ending in `.c` in the current directory to the directory `code`:

```
(for-each
 (lambda (f)
  (rename-file
   f
   (string-append "code/" f)))
 (file-match "." #f "*.c"))
```

The scsh code is programmatic: we generate a list of files (`file-match`) in the current directory (`"."`) excluding hidden dotfiles (`#f`) that end in `.c`, and then we iterate through them (`for-each`) renaming each one to a carefully reassembled name in a subdirectory. Compare with the shell snippet:

```
mv *.c code/
```

How is the shell so concise? Two factors contribute: the `mv` function is variadic, and the pathname expansion stage automatically 'iterates' through the matching files.

To be fair, scsh (and Scheme in general) has some of the features one might want: the `apply` function allows for variadic interfaces, and *quasiquoting* allows the progammer to easily mix code and data in way not dissimilar to word expansion. One could write the bulk file move above in maybe less idiomatic scsh as:

```
(run (mv ,@(file-match "." #f "*.c") code/))
```

Here `,@` is the 'unquote-splicing' operator in quasiquotation. Unquote-splicing splices its argument into the quasiquoted list: after computing the list of matching files, the resulting list is flattened into the list given to `run`. Quasiquoting has a non-splicing insertion, as well. For example, we could abstract out the target as follows:

```
(define (bulk-move-c tgt)
 (run (mv ,@(file-match "." #f "*.c") ,tgt)))
```

Here `,` is the 'unquote' operator. It adds what follows as-is into the list, without splicing. Since the `run` primitive expects a valid command-line to run, the result of quasiquotation here had better be a list of plain strings.

Quasiquoting gets us closer to something we might interactively write, but we're still a ways away from an interactive shell:

(1) The default ought to be running commands, while scsh requires one to type `run` before every command.
(2) Having pathname expansion with `*` greatly simplifies enumerating files.
(3) Quasiquoting requires the user to explicitly decide between unquote and unquote-splice at each inclusion.
(4) Word expansion supports concatenation automatically: if we wanted to make sure `tgt` ends with a slash in `bulk-move-c`, we must write `... ,(string-append tgt "/")`, whereas in the shell, we simply tack a `/` on the end.

In summary, scsh is unsuitable for interactive use not because it's missing `.login`, but because it lacks the concision the shell gains by use of word expansion.

I should be very clear: the programmatic features in scsh and Shill are excellent, and I too seek out similarly well-structured interfaces when programming. My point is rather that there's a dovetail effect that makes the POSIX shell particularly good at interactive work: on the one hand, we have standard POSIX utilities with variadic interfaces; on the other hand, word expansion in the shell gives us a lightweight, concise interface for specifying multiple arguments.

4.1 REPLs and interactivity

While a variety of languages offer REPLs for interactive exploration, two classes of languages are particularly good for both interactive use and programming: scientific computing platforms, like Matlab and R; and dynamic languages in the Lisp tradition, like Racket and Clojure. Neither of these use word expansion, yet they manage to be throughly interactive. How?

Scientists use workbenches like Matlab and R for interactive/exploratory use, ranging from one off commands to, say, generate a graph all the way to longer workflows that are then transitioned to more permanent scripts and programs. I suspect that the following factors contribute: a restricted set of datatypes of interest (largely scalars, vectors, matrices, and data frames); good defaults for visualization (plots and graphs); and large operations bundled up so that a single command includes a great deal of computation (e.g., BLAST, SVD, PCA, and regression libraries). Some of the exploratory nature of these workbenches may be come from their visualizations: it's very easy for a scientist to inspect partially constructed models. I see a cognate in the shell programmer's habit of echoing commands before actually running them. Some of the interactivity may also come from training: if scientists are taught to use these workbenches to explore, then the workbenches develop a *reputation* for being good for interactivity and exploration whether or not they are good for the task.

Lisp family languages like Racket and Clojure support a great deal of interactivity: it's quite common to iteratively add definitions to a file of code during interactive work. That is, programming is a sort of cycle of "explore, find and commit to a definition, explore again, revise or find a new definition". Such a cycle is qualitatively different from shell programming, which is perhaps more about processes (scripting a particular sequence of events) than definitions (designing and manipulating a particular data structure). To put it differently, these interactive sessions in Lisp-y languages are about processes for new data structures, while shell scripts tend to deal with only one data structure—the filesystem. Scsh is an example of a Lisp-like language that is well and truly about manipulating the filesystem, but it is substantially less interactive than the shell (see Section 4, above).

I list these examples of REPLs—scientific workbenches, Lisp-like languages—to make it clear that by no means does the shell have a monopoly on interactive work. But each of these examples is either narrow in scope (scientific workbenches) or not about manipulating the filesystem (Lisp-like languages).

4.2 Shell-like libraries

The Plumbum library for Python and the turtle library for Haskell offer 'shell combinators' [2, 8]. Programmers can reflect shell utilities into language-level functions. Neither is really ideal for interactive use, but both do a good job of embedding shell-scripting DSLs in a more general programming language. I omit further consideration of turtle, since it doesn't aim to be interactive:

> The turtle library focuses on being a "better Bash" by providing a typed and light-weight shell scripting experience embedded within the Haskell language.

The following examples are taken from the Plumbum documentation, and are meant to represent an interactive Python session with Plumbum. First, overloaded operators allow for a shell-like syntax:

```
>>> # compose a shell-like pipe
>>> chain = ls["-l"] | grep[".py"]
>>> # expose the Plumbum representation
>>> print chain
C:\Program Files\Git\bin\ls.exe -l
 | C:\Program Files\Git\bin\grep.exe .py
>>>
>>> chain() # run the pipe
'-rw-r--r--  1 sebulba Administ
  0 Apr 27 11:54 setup.py\n'
```

Once utilities can be invoked like normal functions, one can use built-in Python features like `apply`, `*args`, and `**kw` to support variadic interfaces. The syntax is not quite as spare as that of the POSIX shell, though it's considerably more concise than standard Python idioms for opening processes, like `popen`. Plumbum supports some level of nesting of commands: one can invoke the reflected `ssh` command with a Plumbum pipe itself; the following will connect to `somehost`, then connect to `anotherhost`, and then find files that end in `.py`:

```
>>> ssh["somehost",
    ssh["anotherhost", ls | grep["\\.py"]]]
...
```

Plumbum's abstractions ultimately fail for commands, though: "command nesting works by shell-quoting (or shell-escaping) the nested command" [8]. That is, Plumbum cannot avoid relying, at some point, on the string-based, word-expansion approach of the shell. Plumbum's abstractions seem particularly successful for paths: globbing is explicit, and paths are kept as objects, rather than strings—doing so allows for much more graceful handling of lists of paths than in the shell, where field splitting interacts poorly with spaces in filenames. Relatedly, the Sh library for Python is similar to Plumbum (and inspired Plumbum itself), but aims even less than Plumbum to be a shell replacement. Sh is instead a nicer way to interact with processes in general [6].

Shell libraries like Plumbum and turtle help write scripts, but don't achieve the interactivity of the shell.

5 ASSESSING THE IMPORTANCE OF WORD EXPANSION

The foregoing qualitatively and theoretically examines how word expansion is important for the shell, with my own experience as the sole empirical source. I could instead quantitatively study how the POSIX shell is used in a variety of settings: which features are meaningfully employed by a variety of users when working in the shell? Such a study would bring new forms of evidence to my argument, would complement my approach, and would probably offer other interesting insights into the design of the POSIX shell. I can imagine performing a study in the manner of Whiteside et al. [17]: compare user performance in a variety of modes (the shell; Python or scsh; Python with Plumbum) on the sort of task one would ordinarily perform interactively with the shell (say, The Command Line Murders [16]), breaking users up into groups based on past experience and preference. I suspect that, in general, HCI/UI methods would have interesting ways of phrasing and answering questions about the importance of particular features of the POSIX shell.

6 FIXING AND EXTENDING WORD EXPANSION

I have argued that word expansion is an essential element in the POSIX shell's interactivity: the activities and core abstractions o of the shell demand extensive string manipulation; more things in the shell are operands than operators, and the shell's operators are often variadic; attempts at replacing the shell that leave out word expansion have failed to produce compellingly interactive shells.

Supposing I am correct, and word expansion *is* critical to the shell's interactivity: what can we do to fix the shell, which is undeniably error prone? What features is it missing?

Some popular shells are more (bash [5]) or less (fish [7]) POSIX compliant, extending the POSIX shell with helpful features. For example, bash extends word expansion. Two examples are brace expansion—where `a{b,c}` expands to the two fields `ab` and `ac`—and pattern substitution, where `${x/.c/.o/}` expands to `test.o` when `x` is `test.c`. These extensions are useful, but do nothing to address issues with, e.g., filenames with spaces. Fish's extensions are much more extreme, and with an eye to avoiding errors: they replace the command language with a more 'modern' syntax; some variables, like `PATH`, can range over lists rather than strings, which solves some issues with spaces; they use a different command substitution syntax; they provide automatic shell completion based on parsing manual pages and highlight syntax in the shell based on those completions. While fish's extensions are popular, the fish scripting language does not seem to have the traction of the POSIX shell and does nothing to address existing scripts.

Giger and Wilde [1] add yet another stage of expansion to the shell, extending the `*` and `?` from the POSIX standard's pathname expansion with XPath.

Jeannerod et al. [10] propose using the CoLiS language as a core calculus for studying shell. Their evaluation of string expressions amounts to something akin to word expansion, though their setting is deliberately less complex than what the POSIX standard specifies. Interactive programming seems to be a non-goal for them, since their focus is on analyzing Debian "maintainer scripts" for packages, rejecting programs outside a certain subset of the shell.

Mazurak and Zdancewic [13] describe an analysis for calculating the number of fields that will come out of a given term. More such analyses—perhaps with syntax highlighting à la fish—would surely help identify potential scripting errors.

Word expansion is a critical piece of the shell, dovetailing with the POSIX utilities to offer a concise and powerful interface. Is there some design adjacent to the POSIX shell as it exists that (a) works for many existing scripts, (b) doesn't change the character of the shell so much as to hurt interactivity, but (c) avoids the unpredictability that comes with field splitting?

REFERENCES

[1] Kaspar Giger and Erik Wilde. 2006. XPath Filename Expansion in a Unix Shell. In *Proceedings of the 15th International Conference on World Wide Web (WWW '06)*. ACM, New York, NY, USA, 863–864. https://doi.org/10.1145/1135777.1135916

[2] Gabriel Gonzalez. [n. d.]. Turtle: shell programming, Haskell style. ([n. d.]). https://github.com/Gabriel439/Haskell-Turtle-Library Accessed 2018-02-07.

[3] Michael Greenberg. 2017. Understanding the POSIX Shell as a Programming Language. (2017). OBT.

[4] Stephen José Hanson, Robert E. Kraut, and James M. Farber. 1984. Interface Design and Multivariate Analysis of UNIX Command Use.

ACM Trans. Inf. Syst. 2, 1 (Jan. 1984), 42–57. https://doi.org/10.1145/357417.357421

[5] http://savannah.gnu.org/project/memberlist.php?group=bash. [n. d.]. GNU Bash: the Bourne Again SHell. ([n. d.]). https://www.gnu.org/software/bash/ Accessed 2018-02-05.

[6] https://github.com/amoffat/sh/graphs/contributors. [n. d.]. Sh: Python process launching. ([n. d.]). http://amoffat.github.io/sh/ Accessed 2018-01-31.

[7] https://github.com/fish-shell/fish shell/graphs/contributors. [n. d.]. Fish: the friendly interactive shell. ([n. d.]). https://fishshell.com/ Accessed 2018-01-19.

[8] https://github.com/tomerfiliba/plumbum/graphs/contributors. [n. d.]. Plumbum: shell combinators. ([n. d.]). http://plumbum.readthedocs.io/en/latest/ Accessed 2018-01-31.

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

[10] Nicolas Jeannerod, Claude Marché, and Ralf Treinen. 2017. 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

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

[12] Robert E. Kraut, Stephen J. Hanson, and James M. Farber. 1983. Command Use and Interface Design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '83).* ACM, New York, NY, USA, 120–124. https://doi.org/10.1145/800045.801594

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

[14] Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong. 2014. Shill: A Secure Shell Scripting Language. In *11th USENIX Symposium on Operating Systems Design and Implementation.* USENIX. To appear.

[15] Olin Shivers. 2006. SCSH manual 0.6.7. (2006). https://scsh.net/docu/html/man.html

[16] Noah Veltman. [n. d.]. The Command Line Murders. ([n. d.]). https://github.com/veltman/clmystery Accessed 2018-02-05.

[17] John Whiteside, Sandra Jones, Paula S Levy, and Dennis Wixon. 1985. User performance with command, menu, and iconic interfaces. *ACM SIGCHI Bulletin* 16, 4 (1985), 185–191.

[18] Dennis Wixon and Mark Bramhall. 1985. How Operating Systems are Used: A Comparison of VMS and UNIX. In *Proceedings of the Human Factors Society Annual Meeting*, Vol. 29. SAGE Publications Sage CA: Los Angeles, CA, 245–249.