https://github.com/binpash/popl26-tutorial

Clone this repo!

Working through `src/solution.py`

Easiest to work in docker

Safest too 😅

NB we're shooting for sound-ish

We'll use bash throughout, necessary for part 2

Analyzing Shell Scripts

POPL 2026 // Rennes, France

Tuesday, January 13th, 2026

Michael Greenberg, Konstantinos Kallas,
Nikos Vasilakis, and Evangelos Lamprou

👋 Who we are

POPL 2020	Executable Formal Semantics for the POSIX Shell
EuroSys 2021	PaSh: light-touch data-parallel shell processing
HotOS 2021	Unix shell programming: the next 50 years
ICFP 2021	An order-aware dataflow model for parallel Unix pipelines
PPoPP 2022	Automatic synthesis of parallel unix commands and pipelines with KumQuat
OSDI 2022	Practically Correct, Just-in-Time Shell Script Parallelization
NSDI 2023	DiSh: Dynamic Shell-Script Distribution
HotOS 2023	Executing Shell Scripts in the Wrong Order, Correctly
ATC 2025	The Koala Benchmarks for the Shell: Characterization and Implications
HotOS 2025	From Ahead-of- to Just-in-Time and Back Again: Static Analysis for Unix Shell Programs

👋 Who we are

Some systems papers:

- [NSDI 26, ATC 25 (best paper), NSDI 23, OSDI 22, EuroSys 21 (best paper), HotOS 25, HotOS 23, HotOS 21]

And some PL papers:

- [POPL 20] Executable Formal Semantics for the POSIX Shell
- [ICFP 21] An order-aware dataflow model for parallel Unix pipelines

Recent PaSh Retreat, Fall 25

🚧 Our work today

🐚 shell overview	what is even happening here?	
🐍 parsing POSIX shell	finger exercises in Python with libdash	1, 2, 3
🔣 syntactic analysis	identifying and counting shell features	4
☕ break		
🔍 semantic analysis	identifying pure script fragments	5
🪵 stubs for dynamic work	replace pure fragments with stubs	6
⚠️ JIT instrumentation	replace dangerous commands with try	7, 8

https://github.com/binpash/popl26-tutorial

Clone this repo!

Working through `src/solution.py`

Easiest to work in docker

Safest too 😅

NB we're shooting for sound-ish

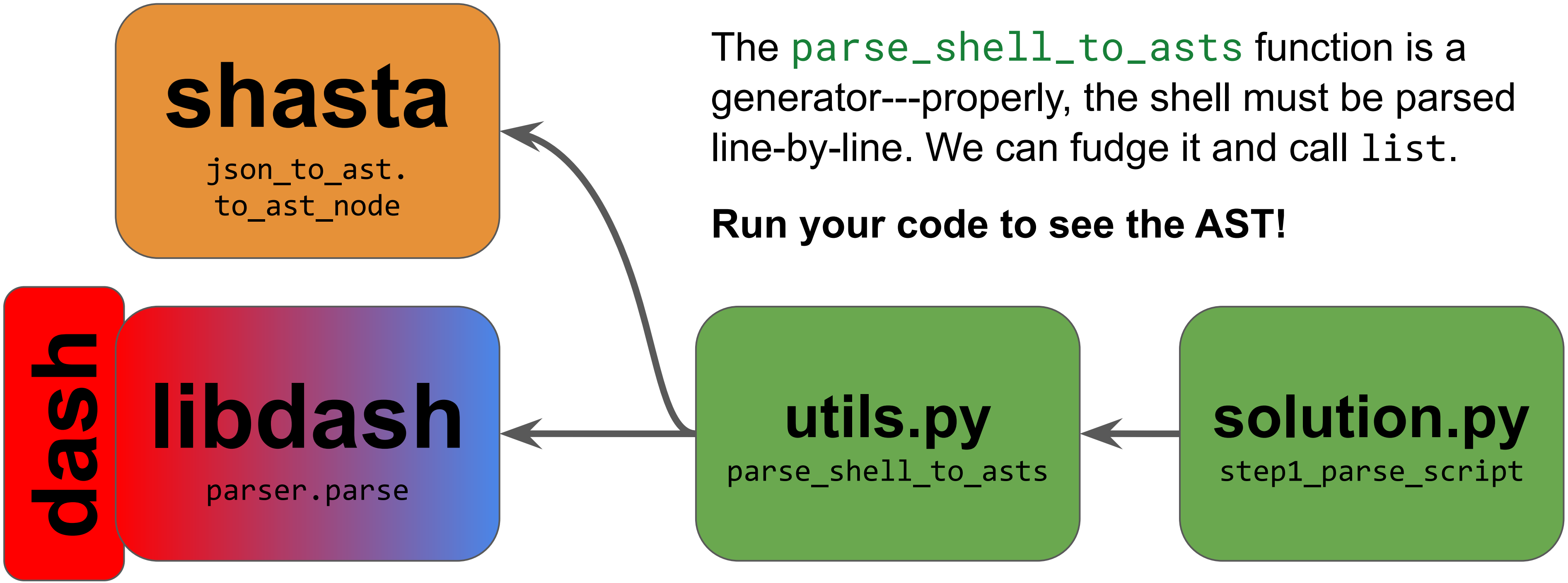
We'll use bash throughout, necessary for part 2

🐍 Step 1: parsing POSIX shell (`step1_parse_script`)

Parse a shell script into a Python object AST!

The `parse_shell_to_asts` function is a generator---properly, the shell must be parsed line-by-line. We can fudge it and call `list`.

Run your code to see the AST!

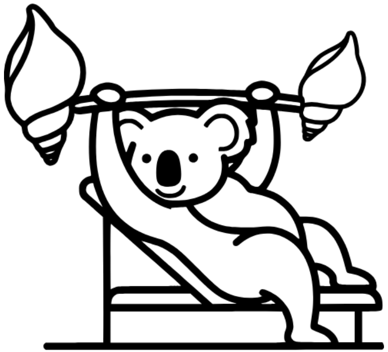


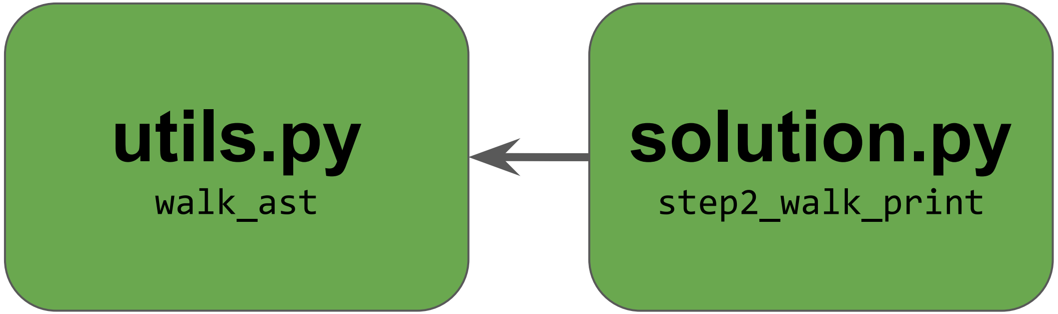
🐍 Step 2: Creating a function that walks the AST

Create an AST visitor! We'll have it work in pre-order, with both visiting and replacement. You have two tasks:

1. Call `walk_ast` with `visit=print`
2. Implement the `AST.CommandNode` case of `utils.walk_ast`

Run your code on a few examples to see how the visitor works.





🐍 Step 3: Pretty-print the script

Render the AST back as a shell script!

Interposing on shell scripts means taking a script as input, manipulating it, and then producing a modified script as output.

Complete the `ast_to_code` function `utils.py`; it takes a list of `AstNode`s and returns a string. It won't look *exactly* the same... play spot the difference with a few scripts!

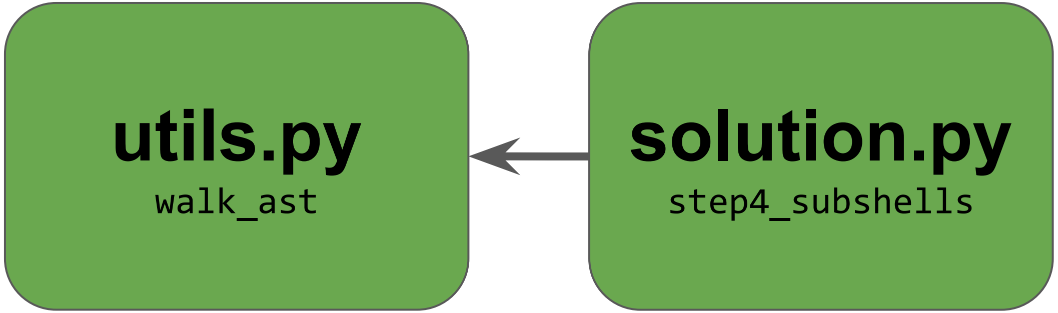
Step 4: Syntactic static analysis of a shell script

Let's write a simple syntactic analysis: how many subshells will a script create?

You might be wondering... which statements create subshells?

There are four ways to create a subshell:

1. asynchronous commands (`... &`; makes 1 subshell)
2. pipes (`... | ... | ...`; makes *n* or *n*-1 [depends on shell!])
3. subshells (`( ... )`; 1)
4. command substitutions (`$( ... )`; 1)



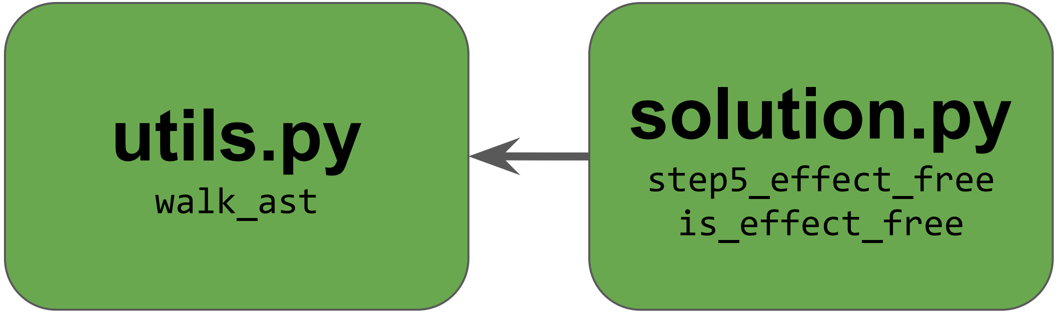
☕ break time! 🥐

🔍 Step 5: Static analysis

Let's build a more serious analysis: which commands can affect the shell's binding? You'll implement a predicate `is_effect_free` using `walk_ast`. What can affect bindings in the shell? We'll use the following:

- function definitions (`fun() { ... }`)
- commands with a non-zero number of assignments (`VAR=val ...`)
- assigning parameter expansion (`${VAR=val}` or `${VAR:=val}`)
- arithmetic expansion (`$((...))`)

Some builtin commands do, too... but let's not worry about that.



🔍🔍🔍 “Analyzing” shell scripts 🔎🔎🔎

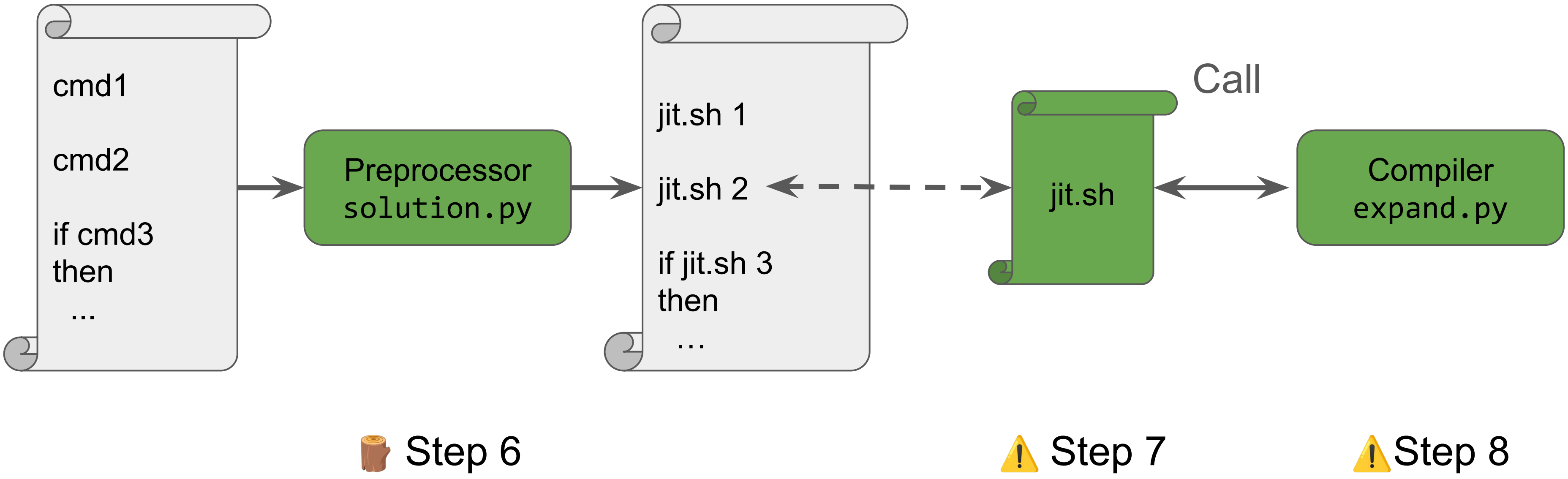
```
cmd=rm; $cmd -rf /
```

```
if gcc -o mystery mystery.c; ./mystery
then cmd=rm; else cmd=echo; fi;
$cmd -rf /
```

```
gcc -o gen_code gen_code.c && ./gen_code >ohno.sh
. ohno.sh
```

runtime is the right time

JIT architecture



Steps 6-8: JIT intervention for safe `rm`

We will work our way up to building a small JIT that interposes on invocations of `rm`.

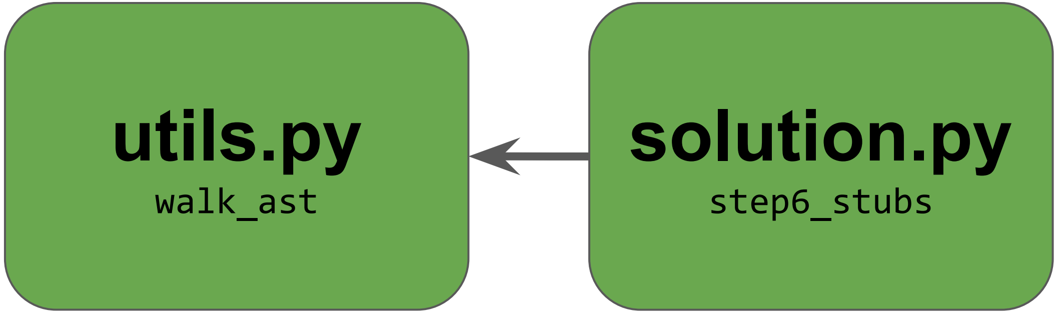
1. **Step 6:** We'll use `walk_ast` to stub out certain commands---the core function of the preprocessor.
2. **Step 7:** We'll define a `jit.sh` that is invoked for each stub---avoiding repeating code during preprocessing.
3. **Step 8:** We'll have our JIT capture the shell's environment and hand it off an expansion routine (`expand.py`), which lets us dynamically detect invocations of `rm` that don't statically appear (e.g., `cmd=rm; $cmd -rf /`).

🪵 Step 6: stubs for dynamic interposition

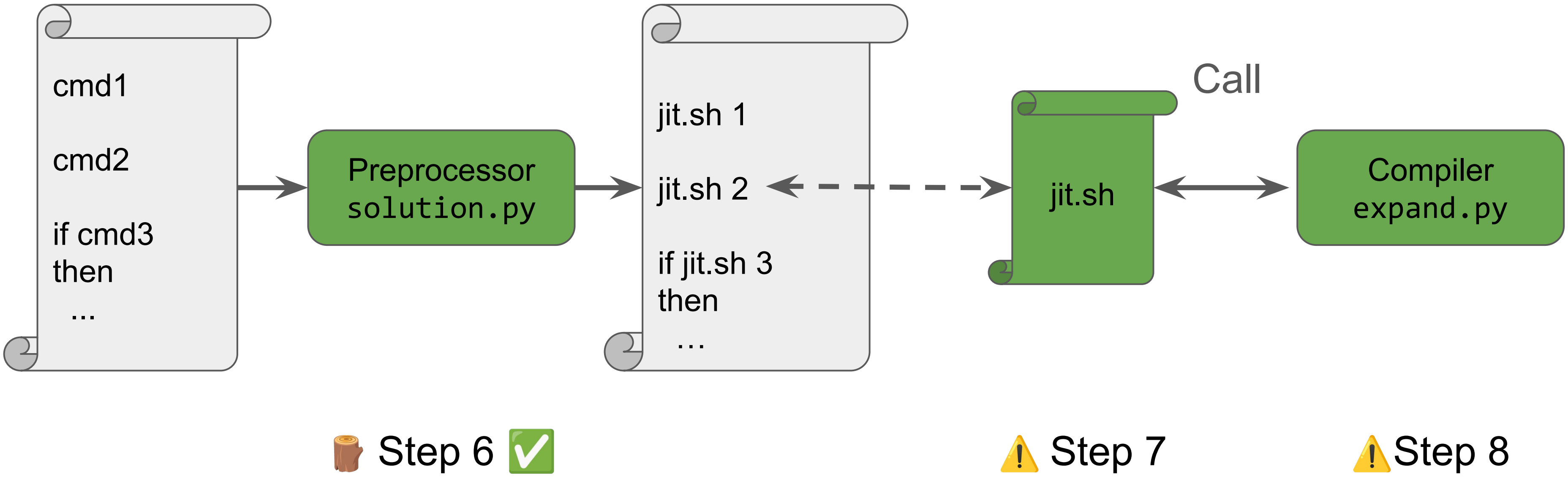
Let’s implement a ‘dry run’ transformation. We’ll alter the script to print out each non-effectful command rather than running it.

Walk the AST with a `replace=...` function that finds nodes that satisfy `is_effect_free`.

For each such fragment, pretty print the code into a ‘stub’ in a known location, and then generate new code that `cat`s that stub out, instead.



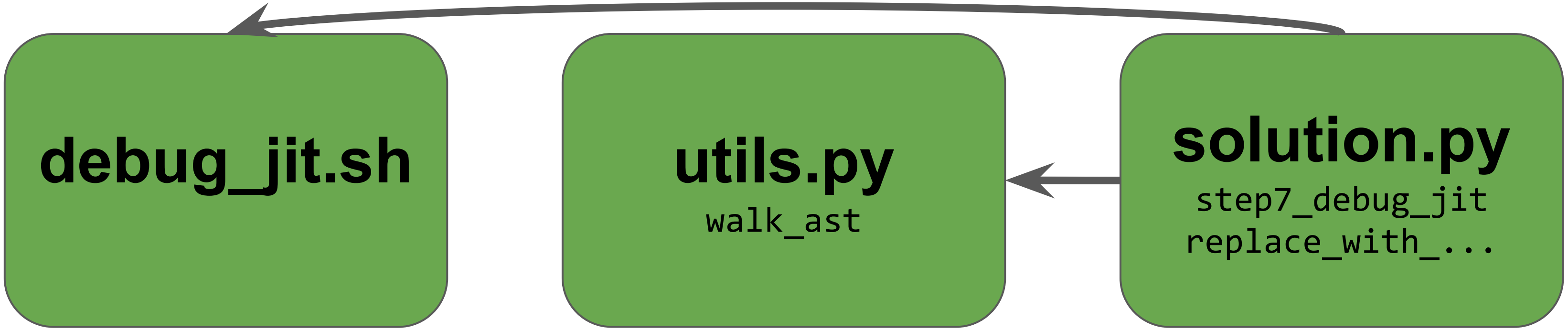
JIT architecture



⚠️ Step 7: Build a simple JIT interposition

Build a JIT script that debugs effect-free fragments *and* runs them, like a minimal `set -x`. You'll need to fill in a bit of `replace_with_debug_jit` so that it generates the JIT invocation of the correct stub:

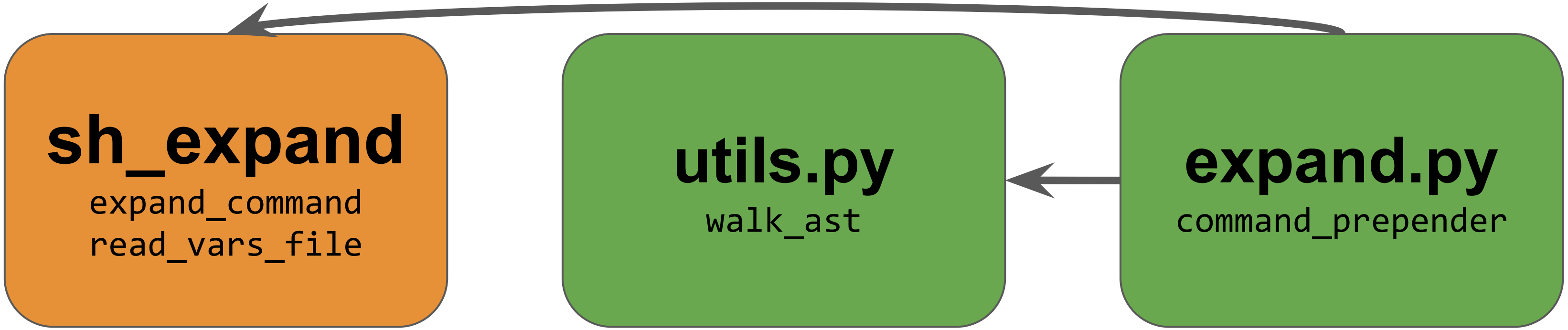
```
JIT_INPUT=/PATH/TO/STUB . src/debug_jit.sh
```



⚠️ Step 8: Interpose on `rm`, just in time

Identify invocations of `rm` just in time, and wrap them in a semi-isolated container using `try`.

Our interposition JIT will identify *every* simple command invocation (with or without effects!) and stub it to hand off to the JIT, which captures the shell environment and hands it to `expand.py`, which can use `sh_expand` to determine which command is being invoked even for, e.g., `cmd=rm; $cmd -rf /`.



⚠️ Step 8: Interpose on `rm`, just in time

The work here is complicated enough that you should do it in two steps.

1. Simply prepend `try` to each `CommandNode`.
2. Implement the optimization.
   a. Uncomment the `deepcopy` call.
   b. Expand the command using `expand.expand_command`.
   c. Extract the executable name using `string_of_expanded_arg`.
   d. If the executable is not unsafe per `unsafe_commands`, then you need not prepend `try` and can return early.

Where do we go from here?

Today: built a proof-of-concept JIT for dynamic intervention.

What moves it beyond a proof-of-concept? PaSh's JIT jumps through more hoops:

env save/restore   perf optimizations   control (`break`, `continue`, `return`, `.`)

What's can *you* do with it?

your analysis here?   your intervention here?   can your work use Koala?

What can *we* do with what you have?

extend Koala?   string reasoning?   filesystem reasoning?   binary analysis?

thank you!

feedback form at
https://forms.gle/q87M5FzXSUwMZnFf6

please let us know what you thought!