GO files lspath - deeper

Below is a two-part answer:

1. A concise but complete walk-through of every source file you pasted – what it does, how it fits into the bigger picture, and the important algorithms/data-flows.
2. Two PlantUML diagrams (a class diagram and a sequence diagram) that you can paste straight into any PlantUML renderer.

PART 1 – CODE EXPLANATION

GLOBAL PURPOSE
“lspath” is a CLI/TUI tool that helps macOS/Linux users understand *why* their PATH looks the way it does.
It can work in two modes:

- Session mode – simply parses $PATH as it exists **right now**.
- Trace mode – starts a **sandboxed** shell (zsh or bash) with a minimal initial PATH, enables `set -x` (xtrace), captures every line that changes PATH, and reconstructs the *exact* file+line responsible for every directory that ends up in PATH.
  The sandbox guarantees that the trace is reproducible and not polluted by the user’s current session.

The code is split into three logical packages:

`internal/model` – pure data structures that every other package imports.
`internal/trace` – how to run the sandboxed shell and how to turn megabytes of “+ file:123>export PATH=…” lines into a concise model.
`cmd/...` (not shown) – glue, TUI, CLI flags, etc.

FILE-BY-FILE

1. model/fileutil.go
   Helper: `GetLineContext(path,line)` → returns a struct with the requested line plus two lines of before/after context. Used when we want to quote the exact line that added a PATH entry.

2. model/icons.go
   Centralised Unicode glyphs used by the TUI/report to keep visuals consistent without pulling in heavy libraries.

3. model/path.go
   All domain types:

4. `PathEntry` – one directory in PATH plus rich metadata (duplicate-of, symlink-target, session-only, remediation advice, diagnostics, …).

5. `ConfigNode` – one shell-startup file (or ghost node if it was *not* executed). Keeps depth (nesting level), order, and indices into the global PathEntry slice.
6. `TraceEvent` – one raw line coming from the trace parser.

7. `AnalysisResult` – final payload sent to the UI/report generator.

8. model/version.go
   Const string injected at build time.

9. trace/analyzer.go (≈ 600 LOC – the brain)
   Three public entry points:

10. `AnalyzeSessionPath(path string)`
    Fast path for "just parse $PATH". Detects duplicates, symlinks, missing dirs, but **no file attribution**.

11. `Analyze(events []TraceEvent, initialPath string)`
    Full reconstruction. Builds the timeline of PATH mutations, keeps attribution (file+line), collapses duplicates, detects symlinks, injects "ghost" ConfigNodes for standard files that were *not* executed (so the user sees the canonical zsh/bash startup order even when some files are missing).
12. `AnalyzeUnified(sessionPath, events)`
    Hybrid: it first runs the trace to get perfect attribution, then overlays the **real** $PATH from the user session.
    Any entry that exists in the live PATH but *was not* in the trace is marked `IsSessionOnly` (typical for `venv`, `npm prefix -g`, Docker Desktop, etc.).
    This gives the user a single list that is both *complete* and *fully attributed*.

Key internal helpers: - `expandTilde()` – `~` → home directory.
- `getPathDescription()` – pretty label such as "(system-wide env)".
- `isImportantConfig()` – standard files that should *always* appear in the flow graph even when they do not touch PATH (zshrc, bash_profile, …).
- `injectMissingNodes()` – inserts not-executed ghost nodes so the final order is the canonical zsh/bash startup sequence. This makes the diagram easy to read for beginners.
- `GenerateReport()` – plain-text or verbose report with icons, remediation hints, statistics, flow graph, etc.

1. trace/executor.go
   `RunTrace(shell, initialPath)` – starts `zsh -xli -c exit` (or bash equivalent) with:

2. $PATH stripped and reset to the constant `SandboxInitialPath` (`/usr/bin:/bin:/usr/sbin:/sbin`).

3. $PS4 set to a deterministic string so the regex in the parser works.
   Returns an `io.ReadCloser` of **stderr** (that is where xtrace writes).
   The caller (parser) streams this in real time.

4. trace/parser.go
   `Parser` owns a regex that scrapes every xtrace line:
   `+ file:line>command` (zsh) or `+file:line>command` (bash).
   If the command contains `PATH=` it extracts the new value and emits a `TraceEvent`.
   The channel of events is fed into `Analyzer.Analyze()`.

5. trace/shell.go
   Tiny interface `Shell` plus two implementations `ZshShell` and `BashShell`.
   `DetectShell()` looks at the user's login shell (or explicit flag) and returns the correct implementation so the rest of the code is shell-agnostic.

IMPORTANT ALGORITHMS / DATA-FLOWS

1. Duplicate detection
   Two maps while we iterate once over the final slice:
2. `seen[value]index` – canonical path (symlinks resolved).

3. `resolvedPaths[target]index` – only used when the current entry itself *is* a symlink.
   Complexity: O(n) and single pass.

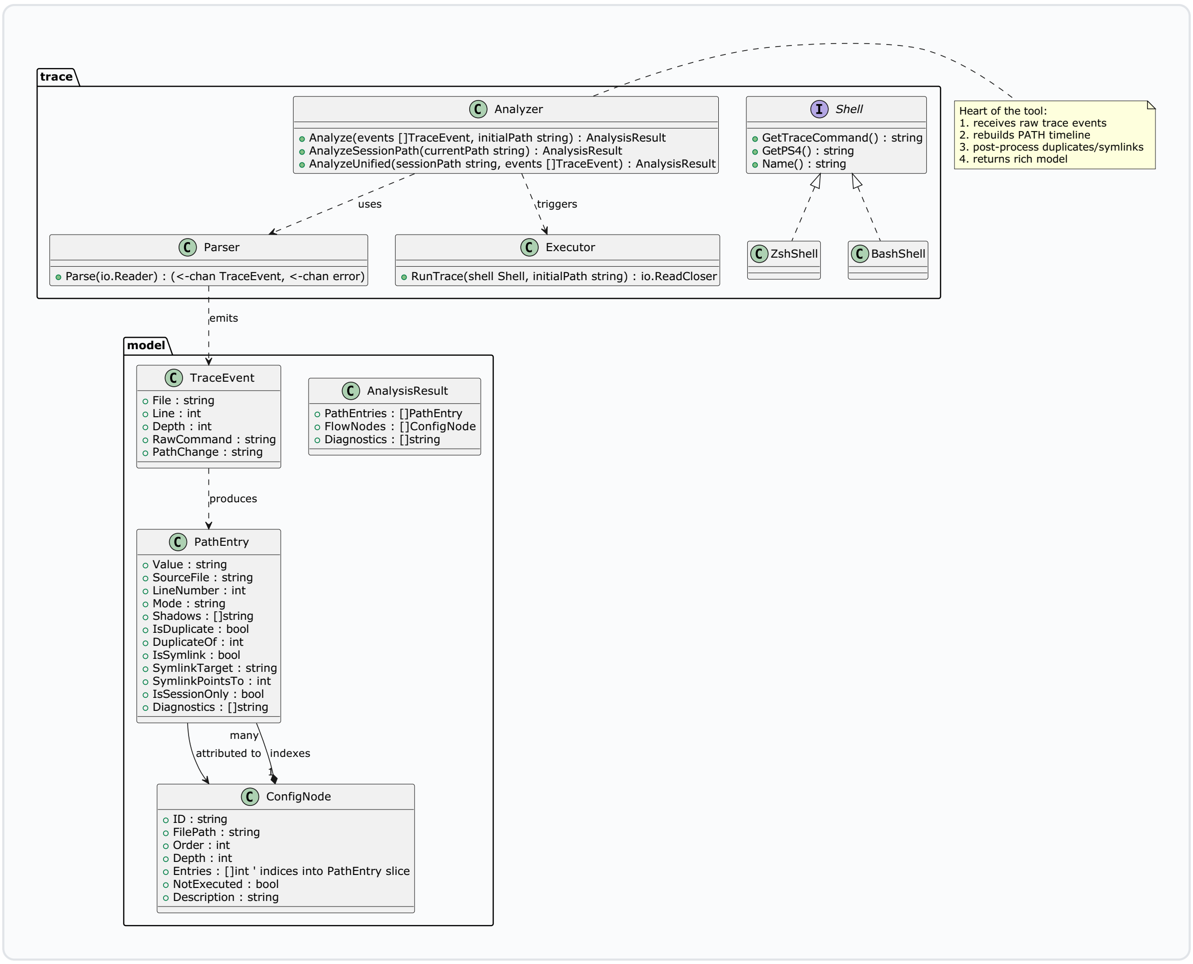
4. Symlink handling
   We `lstat` every PathEntry.
   If `Mode()&os.ModeSymlink != 0` we `readlink`, absolutise relative targets, clean the path, store in `SymlinkTarget`.
   Later we check whether `SymlinkTarget == Value` of another *earlier* entry; if so we set `SymlinkPointsTo=thatIndex` and print a friendly message instead of treating it as a duplicate directory.

5. Eval context attribution
   When the trace sees
   `eval "$(something that eventually exports PATH)"`
   we remember the line of the *eval*; any subsequent PATH change coming from the **same file** is attributed to the eval's line number instead of the later line where the assignment actually happens.
   This gives the user the *correct* file:line to look at.

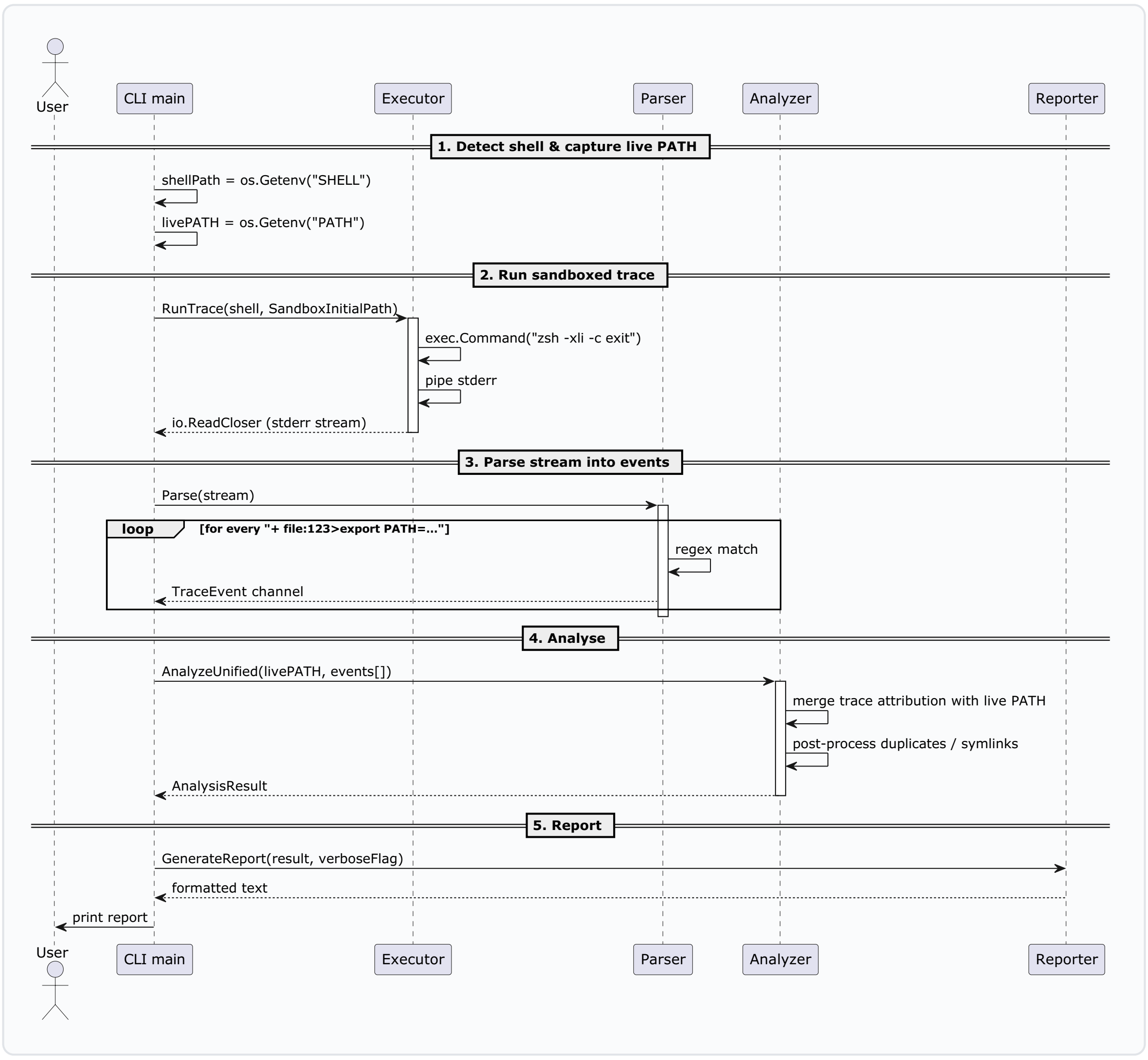
6. Ghost / missing-node injection
   Canonical startup order (zsh):
   `/etc/zshenv → ~/.zshenv → /etc/zprofile → ~/.zprofile → /etc/zshrc → ~/.zshrc → /etc/zlogin → ~/.zlogin`
   If the trace *didn't* execute one of these we still insert a `NotExecuted` node so the final diagram is educational.

PART 2 – PLANTUML DIAGRAMS

CLASS DIAGRAM – main domain objects only (omits trivial helpers)



SEQUENCE DIAGRAM – typical unified workflow (what happens when the user types `lspath --unified`)


User
CLI main
Executor
Parser
Analyzer
Reporter
1. Detect shell & capture live PATH
shellPath = os.Getenv("SHELL")
livePATH = os.Getenv("PATH")
2. Run sandboxed trace
RunTrace(shell, SandboxInitialPath)
exec.Command("zsh -xli -c exit")
pipe stderr
io.ReadCloser (stderr stream)
3. Parse stream into events
Parse(stream)
loop
[for every "+ file:123>export PATH=..."]
regex match
TraceEvent channel
4. Analyse
AnalyzeUnified(livePATH, events[])
merge trace attribution with live PATH
post-process duplicates / symlinks
AnalysisResult
5. Report
GenerateReport(result, verboseFlag)
formatted text
print report

Copy-paste either block into any PlantUML renderer (PlantUML server, IntelliJ plugin, VS-Code extension, etc.) to obtain the diagrams.

Metadata

Creator: admin
Last Modified: 2026-01-15 02:57
Created: 2026-01-15 02:55
Source: GitHub - abulka/lspath @ main
Selected Files:

- `internal/`
- `internal/model/`
- `internal/model/fileutil.go/`
- `internal/model/icons.go/`
- `internal/model/path.go/`
- `internal/model/version.go/`
- `internal/trace/`
- `internal/trace/analyzer.go/`
- `internal/trace/executor.go/`
- `internal/trace/parser.go/`
- `internal/trace/shell.go/`

AI Service: gituml-ai (kimi-k2-0905-groq)
Tokens Used: 16,799
Processing Time: 19.2 seconds
Description: my latest project
Visibility: Private