Kein Performanceunterschied bei Inversion

1. AST-Normalisierung & SSA (Control Flow Graphs)

V8 (über den Turbofan-Optimierungs-Compiler) führt deinen Code nicht zeilenweise so aus, wie du ihn schreibst:

- **Control Flow Graph (CFG):** V8 wandelt sowohl verschachtelte Blöcke als auch Early Returns in dieselbe abstrakte Repräsentation um (Static Single Assignment / SSA Form).
- **Pfad-Reduktion:** Ein verschachtelter Baum `if (A) { if (B) { if (C) { ... } } }` wird im Zwischencode auf denselben Entscheidungspfad reduziert wie `if (!A) return; if (!B) return; if (!C) return;`.
- **Identischer Maschinencode:** Sobald Turbofan den Code optimiert, erzeugen beide Varianten exakt dieselbe Sequenz aus bedingten Sprungbefehlen (`test`, `jnz` / `jz`) auf Assemblerebene.

2. Hardware-Ebene: CPU Branch Prediction

Unabhängig von der JS-Engine entscheidet die Hardware über die Geschwindigkeit von Verzweigungen:

- Wenn Bedingungen wie `isLoggedIn` stabil sind (z. B. immer `true`), lernt der **Branch Predictor** der CPU das Muster nach wenigen Durchläufen.
- Die CPU führt den Zweig spekulativ ohne Pipeline-Stall aus (Branch Prediction Penalty = 0 Takte).
- Ob der nicht genommene Zweig am Ende der Funktion (`nested`) oder direkt hinter dem Check (`early return`) liegt, macht für die Ausführungszeit der CPU keinen Unterschied.

3. Warum misst man im Firefox (SpiderMonkey) manchmal Unterschiede?

SpiderMonkey (Firefox) und V8 (Chrome) verfolgen unterschiedliche Strategien bei der Bytecode-Erzeugung und den JIT-Stufen:

Aspekt	Chrome (V8)	Firefox (SpiderMonkey)
Pipeline	Ignition (Interpreter) → Sparkplug → Maglev → Turbofan	C++ Interpreter → Baseline Interpreter → Baseline Compiler → WarpMonkey
Bytecode-Layout	V8 optimiert Jump-Targets früh im Bytecode-Generator; Basic Blocks werden linear angeordnet.	SpiderMonkey behält in frühen Phasen oft ein Bytecode-Layout bei, das näher an der Quellcode-Struktur liegt.

Bailout / OSR	Sehr aggressives Inlining und Dead-Code-Elimination im `Maglev` / `Turbofan` -Layer.	`WarpMonkey` nutzt Transpilation über CacheIR; je nach Verschachtelungstiefe können Scope- und Frame-Handling im Baseline-Tier minimal variieren.

In **nicht-hochoptimiertem Code** (z. B. Skripte, die nur wenige Male laufen und im Interpreter bzw. Baseline JIT verbleiben):

- Verursacht tiefe Verschachtelung in manchen Engines zusätzlichen Overhead beim Verwalten von Lexical Environments/Scopes auf dem Stack.
- Early Returns erlauben es dem Interpreter, den aktuellen Stack-Frame schneller abzubauen, ohne tiefer liegende Scope-Hierarchien zu durchlaufen.

Sobald der Code jedoch "heiß" läuft (nach einigen tausend Iterationen), eliminieren sowohl Turbofan als auch WarpMonkey diesen Unterschied vollständig.