I can only work for like 1 hour daily, change the road map accordingly

Here's a compact, one-hour-per-day roadmap that keeps a steady cadence and ensures visible GitHub progress. The plan compresses each feature into 3–4 tightly scoped, test-first micro-steps so that every session ends with a passing state, following incremental compiler methodologies designed for small sittings.

Quick answer

Work one focused hour daily on a tiny language compiler, with each session producing a small, testable increment: lexer → parser → AST → semantics → codegen → functions, tagging mini-releases weekly. This mirrors incremental "tiny compiler" practices that deliver a working subset at each step and fit time-limited schedules.[1] [2] [3] [4]

Pacing principles

- One-sitting increments: Design every task to be completable and testable within a single hour to maintain momentum. This incremental compiler approach explicitly targets time-limited learners.[1]
- Always shippable: End each hour with passing tests and a commit; prioritize the smallest vertical slice that runs end-to-end for the current subset.[1]
- Grow in tiny steps: Start with integers → unary → binary → variables → control → functions, as used in popular tutorials and courses.[2] [3] [4]

Week 0: Bootstrap (1 hour/day)

- Day 1: Create repo, README (scope, target: x86-64 or LLVM IR), folder layout, CLI stub to read source from file/stdin, and minimal CI. Commit. This matches toy-compiler course setups emphasizing make/build/CI early.[3] [4]
- Day 2: Draft minimal language v0: ints, + - * /, parentheses, semicolon-terminated statements; write 3–5 example programs and expected behavior. Commit fixtures. Incremental plans start from arithmetic.[2]
- Day 3: Add test harness for golden tests (token streams, AST dump placeholders); wire a test script in CI. Commit. Small, verifiable steps are core to the methodology.[1]

Week 1: Lexer in hour-sized slices

- Day 4: Tokens for integers, + - * / (no identifiers yet), whitespace skipping; unit tests. Commit. Lexing is first stage in compact tutorials.[5] [6]
- Day 5: Add parentheses and semicolons; error on unknown char with line/col. Commit diagnostics. Robust lexer diagnostics are emphasized in tutorials.[6]
- Day 6: Add identifiers and equals to prep for variables later; keep unused for now. Commit. This anticipates semantic/name binding.[6]
- Day 7: Optional comments (// to EOL) and final lexer refactor; tag v0.1. Commit. Coursework often includes stripping comments early.[7]

Week 2: Parser + AST, tiny increments

- Day 8: Recursive-descent for numbers and plus/minus expressions (no precedence yet beyond left-assoc); build minimal AST nodes. Commit. Toy compilers start with simple expressions.[4] [3]
- Day 9: Add multiply/divide and precedence (factor/term/expr). Tests for precedence and errors. Commit. Standard parsing progression.[5] [6]
- Day 10: Parentheses and unary minus; malformed input tests. Commit. These are typical early parser features.[5] [6]
- Day 11: Statement list with semicolons; parse a program of multiple expressions. Commit an AST dump tool.[3] [4]
- Day 12: Add simple error recovery (sync at semicolon) and tag v0.2. Commit. Parser error handling is a key phase goal.[6]

Week 3: Semantics in bite-size steps

- Day 13: Symbol table scaffold; add let statements (let x = 3;). Resolve identifiers. Commit minimal name resolution. Semantic binding is central at this stage.[5] [6]
- Day 14: Type checks for int ops; produce type errors for invalid uses. Commit errors and tests. Tutorials stress type checking as a distinct phase.[6] [5]
- Day 15: Constant folding for pure integer subtrees. Commit before/after AST snapshots. Small optimization fits a single session.[6]
- Day 16: Assignments (x = x + 1;), use-before-def diagnostics. Commit. Name rules round out basics.[6]
- Day 17: Documentation hour: README explaining semantics, error examples; tag v0.3. Commit. Clear specs mirror course expectations.[5] [6]

Week 4: Codegen step-by-step

- Day 18: Decide backend: x86-64 assembly assembled by system toolchain, or LLVM IR with lli. Wire pipeline to emit a minimal main that returns an integer literal. Commit. This mirrors toy compiler assembly-first workflows.[4] [3]

- Day 19: Emit arithmetic expressions into registers/stack; run “1+2*3;” end-to-end. Commit sample outputs. Course notes show assemble/link loops.[3] [4]
- Day 20: Stack slots for variables; load/store for let and assignment. Commit tests comparing program outputs.[3]
- Day 21: If/else lowering: compare, conditional branches; treat nonzero as true. Commit. Control lowering is typical here.[6]
- Day 22: While loops with labels and jumps; tag v0.4. Commit runnable examples.[4] [3]

Week 5: Functions in micro-steps

- Day 23: Function definitions (no calls yet): prologue/epilogue and return; compile a single function + main that calls none. Commit. Calling conv basics begin here.[2]
- Day 24: Function calls and return value; pass args via stack or simple register set; add recursion test. Commit factorial. Sandler’s series highlights call mechanics as a key learning step.[2]
- Day 25: Block scopes and nested symbol tables for locals; shadowing rules. Commit. This rounds out lexical scoping.[6]
- Day 26: Arity/type checks for calls; error diagnostics; tag v0.5. Commit. Validation improves UX.[2] [6]

Week 6: Polish within one-hour blocks

- Day 27: Booleans, relational ops, and short-circuit &&/|| lowered to branches; extend type checker. Commit. Common late-stage additions. [6]
- Day 28: Stdlib shim: external print or putchar linkage; show link step with system toolchain. Commit I/O examples. This surfaces linking realities.[7]
- Day 29: Diagnostics polish: caret spans and hints; add a common error guide in README. Commit. User-facing errors matter.[6]
- Day 30: Language tour in README: grammar, examples, how-to-run, CI badges; tag v1.0. Commit. Course roadmaps encourage complete docs and reproducibility.[4] [3]

Daily one-hour template

- 0–10 min: Write 1–3 tests for the next micro-feature. This aligns with the “one-sitting, test-first, expand subset” method.[1]
- 10–40 min: Implement the smallest slice to make tests pass; keep scope narrow. Micro-steps ensure a working compiler every day.[1]
- 40–55 min: Refactor lightly without breaking tests; update docs/snippets. Iterative refactoring is part of the cycle.[1]
- 55–60 min: Commit with concise message; push; open a short checklist issue for tomorrow. Consistent small commits mirror toy-compiler coursework discipline.[3] [4]

What to read alongside (optional 1–2 sessions/week)

- Follow an incremental blog series for motivation and practical patterns; Nora Sandler's "tiny to real" C-compiler journey exemplifies layering features while keeping scope tight.[8] [2]
- Skim a compact "how compilers work" overview before a new phase to anchor concepts of lexing, parsing, semantics, codegen.[5] [6]
- For confidence in the hour-a-day approach, note the incremental paper's explicit design for time-limited settings.[1]

Release cadence

- v0.1 (end Week 1): Lexer complete with tests.[5]
- v0.2 (end Week 2): Parser + AST dump working.[4] [3]
- v0.3 (end Week 3): Semantics with variables and constant folding.[6]
- v0.4 (end Week 4): Codegen for expressions, vars, control; runnable binaries.[3] [4]
- v0.5 (end Week 5): Functions and calls; recursion passes tests.[2]
- v1.0 (end Week 6): Language tour, diagnostics, basic stdlib/linking published.[7]

Tips for staying within one hour

- Pre-slice tasks: Open issues that are so small they fit in one sitting, matching the "every step yields a working subset" rule.[1]
- Prefer simplest backend first (plain assembly with gcc/clang) before LLVM IR to reduce overhead, as many toy compiler courses do.[4] [3]
- Keep failing tests few and targeted to avoid scope creep; grow coverage gradually like the incremental scheme.[1]

⁂

1. http://scheme2006.cs.uchicago.edu/11-ghuloum.pdf
2. https://norasandler.com/2017/11/29/Write-a-Compiler.html
3. https://www.cs.ucf.edu/~gazzillo/teaching/cop3402spring24/lectures/03_toy_compiler.html
4. https://cop3402fall20.github.io/lectures/03_toy_compiler.html
5. https://dev.to/earnstein/how-compilers-work-a-step-by-step-guide-26c6
6. https://www.geeksforgeeks.org/compiler-design/compiler-design-tutorials/
7. https://www.geeksforgeeks.org/c/compiling-a-c-program-behind-the-scenes/
8. https://www.linkedin.com/pulse/we-speak-your-language-nora-sandler-raincode-labs
9. https://hourofcode.com
10. https://www.designgurus.io/answers/detail/is-1-hour-a-day-enough-to-learn-coding
11. https://williamcareyuniversity.co.in/can-one-hour-a-day-truly-make-you-a-coding-pro
12. https://hourofcode.com/learn
13. https://www.youtube.com/watch?v=ZzaPdXTrSb8

14. https://www.cs.ucf.edu/~gazzillo/teaching/cop3402spring24/projects/toycompiler.html
15. https://www.reddit.com/r/ProgrammingLanguages/comments/1lu9sjd/has_anyone_read_writing_a_c_compiler_by_nora/
16. https://www.reddit.com/r/learnprogramming/comments/relaxs/is_one_hour_a_day_of_learning_programming_enough/