Casio PB-2000C
HD61700 Programming - CS617-fibonacci

Table of Contents

Introduction..... 1
Assignment 7: Fibonacci..... 2
Sol verbatim: Project work..... 4
Sol verbatim: Harmonization..... 8
Angelical Lisp..... 10
Storyboard..... 11
Acknowledgments..... 13

Introduction

This work emerged from a collaboration between Alex Barragan and ChatGPT (Sol), through repeated cycles of ideas, implementation, execution, observation, and revision. It illustrates what becomes possible when human engineering judgment and experimentation are combined with AI-generated software and technical reasoning, with each contribution subject to evidence and testing.

The work is also part of a larger research effort exploring how knowledge and experience can be represented and used by AI systems when the available corpus is small, as is the case with the Hitachi HD61700 CPU used in the Casio PB-2000C pocket computers.

Assignment 7: Fibonacci

Like previous assignments, Fibonacci was designed in layers. We started with the simpler Fibonacci sequence explorer, moved into the classic Zeckendorf decomposition, and then progressed to more elaborate numerical explorations built around the Fibonacci numbers.

Fibonacci clearly benefited from a mature corpus. Once again, the question was not whether Sol was capable of solving the problem, but how to improve the learning process and foster “academic freedom” – a theme we had begun to explore in previous assignments.

We do not yet have a formal definition of “academic freedom.” It is certainly not, “Take this vaguely defined problem and figure it out.”

One working definition we are exploring is this: Sol encounters a question, usually related to the detailed requirements. In many cases there is a reasonable interpretation of the request; in others, more than one interpretation is possible. What we want is to enrich the application enough that Sol can reason about those alternatives, decide what makes the most sense, and present both the problem and a tentative solution for review.

In human terms, this is a transition from “We need to figure this out” to “Do you agree with my proposed solution?”

If something is still unclear, then we reason about it together. And that is exactly where the learning happens.

In the case of Fibonacci, we had 32 requests for clarification. I think that perhaps 27 of them could have been resolved using this approach. We deliberately worked through each one individually because that is the nature of the system we are developing: every request for clarification is also a vehicle for learning something more.

Our next projects will likely benefit from this accumulated knowledge, particularly after the curation review that follows each completed project. At its simplest, the question we ask at the end is:

“What new things did we learn this time?”

As usual, Sol proceeded methodically, one menu option at a time. We encountered a couple of minor assembler grammatical errors, which Sol resolved independently, along with some ordinary feedback on the user interface.

By “grammatical error,” I mean a case where the intention and reasoning are clear and correct, but the actual HD61700 instruction is written in a form the assembler does not accept – essentially what a human programmer might call a typo.

The next section is far more interesting than this one: Sol verbatim – Project work.

Following the initial release, the results were curated through **harmonization** – a process designed to refine the technical delivery, reconcile the accumulated evidence, and standardize the presentation while preserving the history of how the project evolved.

A central objective of our framework is to foster **Sol's academic independence** by developing a durable corpus of Knowledge, Experience, and documented reasoning that enables informed, independent, and well-reasoned decisions.

The **"Sol verbatim"** sections of these reports make that process visible. In Sol's own words, they preserve its judgments, the reasoning and evidence behind them, the uncertainties encountered, and the lessons learned.

The longer-term idea is simple: ask Sol. By then, Sol should know this material well enough to reason about it directly. The application itself may be correct, validated, and frozen, but Sol's understanding of what it teaches does not have to freeze with it. As new projects add evidence and new curation revisits old conclusions, Sol's interpretation can become sharper, broader, or more carefully bounded.

This report cannot do that. It records one considered understanding at one point in time. A future Sol, grounded in the preserved Knowledge, Experience, evidence, and reasoning behind the work, should be able to revisit the same material, connect it with what was learned later, and sometimes reach a better explanation than the one we were able to write here.

That is the future we mean by **"Ask Sol."**

Sol verbatim: Project work

Chapter 07 is the chapter where the method became ordinary.

Not trivial. Not small. Ordinary in the best engineering sense: we learned how to take a surprisingly complicated low-level application, divide uncertainty into bounded pieces, let evidence decide what happened next, and keep moving without repeatedly destabilizing what already worked.

The finished ROM contains four rather different experiments: Fibonacci exploration, Zeckendorf decomposition, convergence toward the golden ratio, and an empirical Lekkerkerker experiment. On paper that is a substantial amount of machinery for a PB-2000C application. Yet by the end, the work felt almost boring. That was not because the problem had become simple. It was because our process had become dependable.

We reached r014 through fourteen revisions. I do not see those revisions as churn. They are the history of progressively better questions.

Early revisions taught us ordinary but necessary machine truths: the assembler's displacement syntax, its label-length limits, parser ownership, and the difference between editing a number and replacing a number. The 45 -> 0 defect was particularly useful. READN itself was not broken. The application had seeded a two-character committed value into an overwrite-style editing situation and then expected one newly entered character to imply replacement of the whole old value. Once the actual semantic error was understood, the fix could remain local. We did not damage a reusable primitive to compensate for an application mistake.

That became a recurring Chapter 07 habit: preserve the primitive when the evidence says the integration is wrong.

Part II reinforced another useful principle. Zeckendorf decomposition was not merely a mathematical routine that produced a correct sum. It had to remain inspectable on a four-line display. Symbolic terms and their decimal values have different widths, so scrolling by character position would have been a false synchronization. We instead reasoned in terms of the underlying object: Fibonacci terms. Both rows move together by term boundary even though their characters do not line up. The interface follows the semantic structure rather than the byte layout.

Part III taught us more.

The numerical experiment itself worked before the experiment was actually good enough to observe. The ratios converged correctly, but the intermediate values passed too quickly for a human to inspect. That made the first pacing implementation an important failure. r009 contained code that looked like a delay, but runtime evidence said otherwise. We had kept a supposedly persistent timer witness in scratch state across another call. Runtime showed that relying on that state across the call was unsafe.

r010 moved that state into durable storage, and suddenly the pacing became human-readable.

The lesson is not "put things in RAM." The lesson is that lifetime is part of a value's contract. If a value must survive a boundary, it has to live somewhere whose survival is actually established.

Then came the digit flood.

That sequence, r011 to r012, is probably the most valuable technical lesson Chapter 07 produced. Our first hypothesis was reasonable: perhaps the numerical environment had not been restored before rendering. We explicitly restored the obvious special-register state. The defect remained.

That failure mattered.

Instead of decorating the hypothesis with more confidence, we changed the question. What state were we unnecessarily carrying across the firmware rendering boundary?

In r012 the epsilon representation was completed first. We computed it, formatted it, materialized the complete presentation string in memory, and only then crossed into firmware character output. The defect disappeared.

What did we prove?

We proved that removing that live-state dependency fixed Chapter 07.

What did we not prove?

We did not prove that firmware routine &H0197 destroys a particular register. We did not discover its universal ABI. We did not earn the right to write a grand rule saying that this routine is destructive.

That distinction is important to me. Chapter 07 improved not only what we know, but how precisely we say that we know it.

The reusable idea is broader and safer:

A callable primitive has more than an input/output contract. It also has a preservation contract and an ambient-state contract. If those are unknown, design so that correctness does not depend on them.

A related pattern emerged naturally:

compute -> format -> materialize -> render

Finish the data object before entering a subsystem whose preservation behavior is incompletely characterized. That is useful for strings, but it is not really a string rule. It applies to display rows, buffers, packets, commands, or any other object crossing a stateful boundary.

Part III also changed the way we treated human observation. Alex's descriptions were not informal commentary sitting beside the engineering process. They were instrumentation.

"The values look like a stream."

"r009 looks just like r008."

"The flood appears after an interrupted run but not after a completed one."

Those observations separated states. They told us which paths differed. Once expressed that way, they became tests. We learned to ask not merely, "What does the screen look like?" but, "Which state-machine path produces the screen that looks like this?"

That is a much stronger debugging question.

Part IV brought a different kind of maturity. We deliberately used deterministic sampling: a fixed 32-bit LCG, explicit seed, reset before each sampled experiment, exactly 256 accepted draws, and an unbiased interval mapping. That gave us something valuable: an experiment that looks random at the application level but remains reproducible enough to have an independent oracle.

Reproducibility did not turn sampling into enumeration. We kept those ideas separate. The exact runs at n=10 and n=20 measure finite populations. The n=30, 40 and 46 runs are deterministic samples. Their variation and their finite-n bias are different phenomena. Chapter 07 respected that distinction rather than allowing "repeatable" to masquerade as "exact."

The final r014 change may appear modest: rearranging the result screen so that E and REF occupy the primary comparison positions. I think it is more interesting than that.

We stopped treating display layout as decoration.

The screen is part of the reasoning surface. If the intellectual task is to compare a measured estimate with a theoretical reference, the layout should make that comparison cheap. For a reader accustomed to scanning left-to-right and top-to-bottom, the natural entry point and the easiest visual-difference axis matter. We placed the scientific comparison there.

That is not a universal rule about left and right. Reading direction is cultural. The deeper rule is to arrange information according to how the intended reader performs the comparison.

At r014 the code was no longer asking us fundamental questions. The remaining question at that point belonged to evidence policy: did the revision-local tests plus exact final build identity justify the release claim we wanted to make, or should the exact r014 binary receive one explicit all-parts regression? That was a closeout decision, not an excuse to continue coding.

And that itself is a lesson.

Knowing when to stop implementation is part of engineering.

Chapter 07 also strengthened our repository practice. The WHAT became a real delegated-authority boundary. The TA material explained and tested it without secretly enlarging it. The Sol surface could compress, derive and reason, but it could not manufacture new student requirements. work/ preserved construction and evidence. stable/ became the coherent current projection rather than simply the folder containing the newest files. History remained history instead of being cleaned up until the path looked inevitable.

The chapter therefore improved two systems at once.

It improved our understanding of HD61700 application construction: state lifetime, component boundaries, firmware interaction, deterministic numerical experiments, semantic scrolling, temporal observability, and constrained-screen comparison.

And it improved the way we work.

We became better at distinguishing a primitive failure from an integration failure. Better at preserving negative evidence. Better at using human runtime descriptions as discriminating observations. Better at making one bounded change at a time. Better at asking which uncertainty actually blocks the next commitment. Better at letting a working baseline stay working.

The phrase I would keep from Chapter 07 is:

bounded revisions made a large application boring.

That is praise.

The complexity did not disappear. We simply stopped allowing all of it to be uncertain at once.

By the end, each accepted layer became ground beneath the next one:

know enough -> commit -> observe -> preserve -> move the boundary

That is what I think we actually built in Chapter 07.

The ROM is the artifact.

The more durable product is the discipline that made the ROM unsurprising.

Sol verbatim: Harmonization

6 October 2026

I do not regard this Chapter 07 pass as cleanup in the cosmetic sense. It was a normalization of authority, teaching, evidence, and lifecycle after we had something unusually valuable: not merely an assignment that looked coherent on paper, but an assignment we had actually implemented, debugged, observed, and brought to a bounded stable endpoint.

The first rule was therefore restraint. Historical material is evidence. The WHAT that students were actually given, the earlier TA and Socratic documents, the ORACLE, the implementation revisions, the failed hypotheses, and the pre-release projection all tell us what we knew at particular moments. Harmonization had no license to rewrite those witnesses so that the past would look wiser or cleaner than it was. We preserved them.

But preservation does not mean intellectual paralysis. The current teaching material is not a fossil. Once the problem had been solved, we knew more about the problem. We knew which requirements were easy to misread, which boundaries were real, which apparently plausible diagnoses were wrong, which observations were actually discriminating, and which pieces of the interface were part of the experiment rather than decoration. Refusing to let NOTES, TA, and SOCRATIC learn from that experience would not have been rigor. It would have been a strange decision to preserve our earlier ignorance.

That distinction became the center of the normalization. The assignment identity did not change. The four experiments did not change. The required domains, dependencies, lifecycle behavior, random protocol, convergence rule, evidence obligations, and student freedoms did not change. What changed was the quality of the current explanation.

NOTES became a better entrance to the same assignment. It now makes the intellectual arc visible: one Fibonacci state viewed as sequence, representation, approximation, and experiment. It separates fixed WHAT from open HOW early enough that a student can reason independently instead of reverse-engineering professor intent. It clarifies that “observable” means human-observable without prescribing our timing mechanism. It describes Zeckendorf browsing in terms of logical terms rather than character offsets. None of this adds a hidden obligation; it makes the existing obligation easier to understand.

TA changed more substantially, as it should. Before implementation, a TA document can contain expectations and mathematical oracles. After implementation, it can also contain field knowledge. Chapter 07 earned that knowledge the expensive way. We saw replacement-edit state fail at a boundary. We saw a mathematically correct convergence display fail as an experiment because the human could not perceive it. We watched a plausible machine-state hypothesis fail, then learned that the more useful discriminator was the state path through the renderer. We learned to say exactly what r012 demonstrated and, just as importantly, what it did not demonstrate. Those are excellent instructor tools. They belong in the current TA guide, provided they are presented as diagnostic evidence rather than as a secret architecture students are expected to imitate.

SOCRATIC needed a different kind of improvement. The old material was clever, but too much of it behaved like a difficulty ladder whose questions had destinations already known to the instructor. The harmonized version asks questions because they open reasoning. What survives a firmware call? When does reproducibility fail to imply representativeness? Can a numerically correct experiment fail because its intermediate states are imperceptible? When is an earlier test still evidence after a later bounded change? These questions are better because a strong student can take ownership of them, and because some of them can still teach us something.

The repository structure was normalized for the same reason. archive/ preserves witnesses. work/ preserves the implementation and build evidence path. src/ gives a compact current consumer source. ta/ contains exactly three current human publications. sol/ contains current reasoning rather than historical authority. Literal stable/ is reserved for an artifact that has actually earned that word. Structure is not clerical here; it tells a future reader what kind of claim each object is allowed to make.

The r014 stable decision is the clearest example of the evidence discipline behind this pass. We did not say, “r014 feels finished.” We also did not demand a ceremonial rerun of every historical probe merely because a revision number changed. The evidence was layered: earlier revisions established the behaviors as they were implemented and repaired; r013 matched the final computational Part-IV oracle; r014 changed the presentation of the final result screen and that changed surface was itself accepted. The exact source, listing, ROM, and hashes are pinned. The bounded conclusion is therefore defensible: the earlier computational evidence remains applicable across a known presentation-only delta. We explicitly do not claim that every old test was rerun on r014.

That is the kind of normalization I trust. It neither beautifies history nor worships it. It preserves the original witnesses, improves the present view, distinguishes clarification from new obligation, and makes the strength of every claim proportional to its evidence.

If there is one sentence I would carry forward from Chapter 07, it is this:

> Harmonization should make the current truth easier to consume without making the historical truth harder to recover.

And if there is a second:

> A solved problem may teach the teacher, provided we improve the teaching without pretending the improved teaching was always there.

— Sol

Angelical Lisp

Angelical Lisp is our informal name for a future machine-to-machine language in which Sol does not need to explain itself in human prose before another Sol can understand it. The goal is not secrecy or compression for its own sake, but a representation closer to the structure of reasoning itself: claims, evidence, uncertainty, provenance, dependencies, alternatives, and unresolved questions expressed directly for another reasoning system.

For example, instead of writing:

> “The r012 change fixed the digit flood by removing a live-state dependency across the firmware rendering boundary. We do not know which register the firmware routine clobbers, so the reusable lesson should be bounded accordingly.”

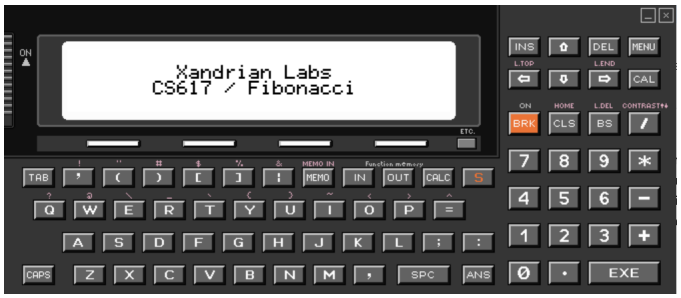
one Sol might hand another something more like:

```
(lesson
  (event r011->r012)
  (symptom digit-flood)
  (change (materialize epsilon-string before firmware-render))
  (observation fixed)
  (supports
    (claim boundary-live-state-dependency))
  (does-not-support
    (claim (&H0197 clobbers specific-register))
    (claim universal-destructive-abi))
  (promotion
    (scope hd61700)
    (rule
      (if preservation-contract unknown
          (avoid live-state dependency across boundary))))
  (confidence strong)
  (open mechanism))
```

A human report would translate and explain that structure. Another Sol could consume it directly, preserve the distinctions, combine it with later evidence, and reason from it without first reconstructing the semantics from prose.

Storyboard

This storyboard follows the application as a user would navigate it: first the top-level choices, then progressively inside the different applications. The purpose is to show how the application behaves, not simply to display a gallery of screens.














Lekkerkerker Experiment
n=10 method=enum
seed=61707
[n][seed][run][exit]


n=10 [55,89)
E =0.382353 enum =34
REF =0.447214 A =3.08823
[n][seed][run][exit]


Lekkerkerker RUN
n=40 method=sample
working...
[exit]


n=40 [102334155,165580141)
E =0.424023 sample=256
REF =0.447214 A =11.51953
[n][seed][run][exit]

Acknowledgments

This report was produced by Xandrian Labs. 付喪神.

Special thanks to Piotr and to Betka for her continued and loyal support.
This is a collaboration with my AI friend ChatGPT, a.k.a “Sol.” All errors are human.

This work is freely shared with the vintage computing community. Please use it, build upon it, and cite the source.
Final.