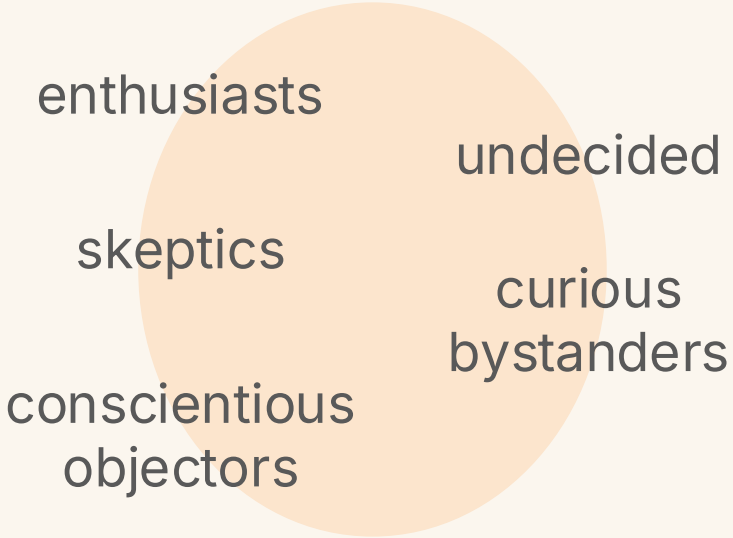
Specifications, Tests, and Audits: the Building Blocks of First-Year CS in an Agentic Era



Kathi Fisler, Brown CS
Joint work with Shriram Krishnamurthi



Context setting



I'm speaking as
- teaching faculty (CS1 and CS2),
- director of undergrad studies,
- CS Ed researcher, and
- moderate Claude user

conscientious objectors out of scope in this talk

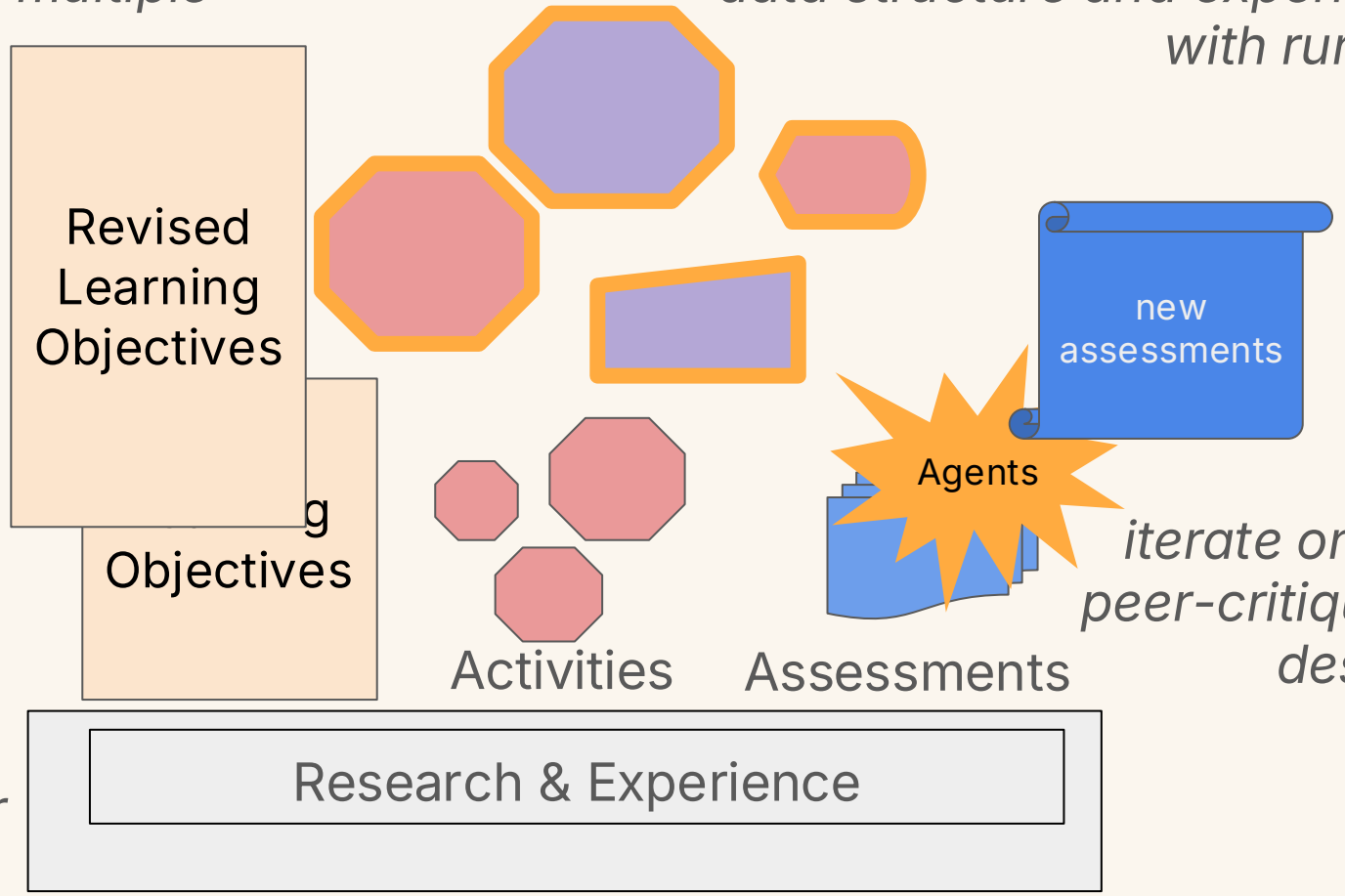
create analytics dashboard that combines multiple datasets

adapt assignment to a different data structure and experiment with runtime

contrast using SQL with manual data queries

develop wrapper to adapt own design to API for autograder

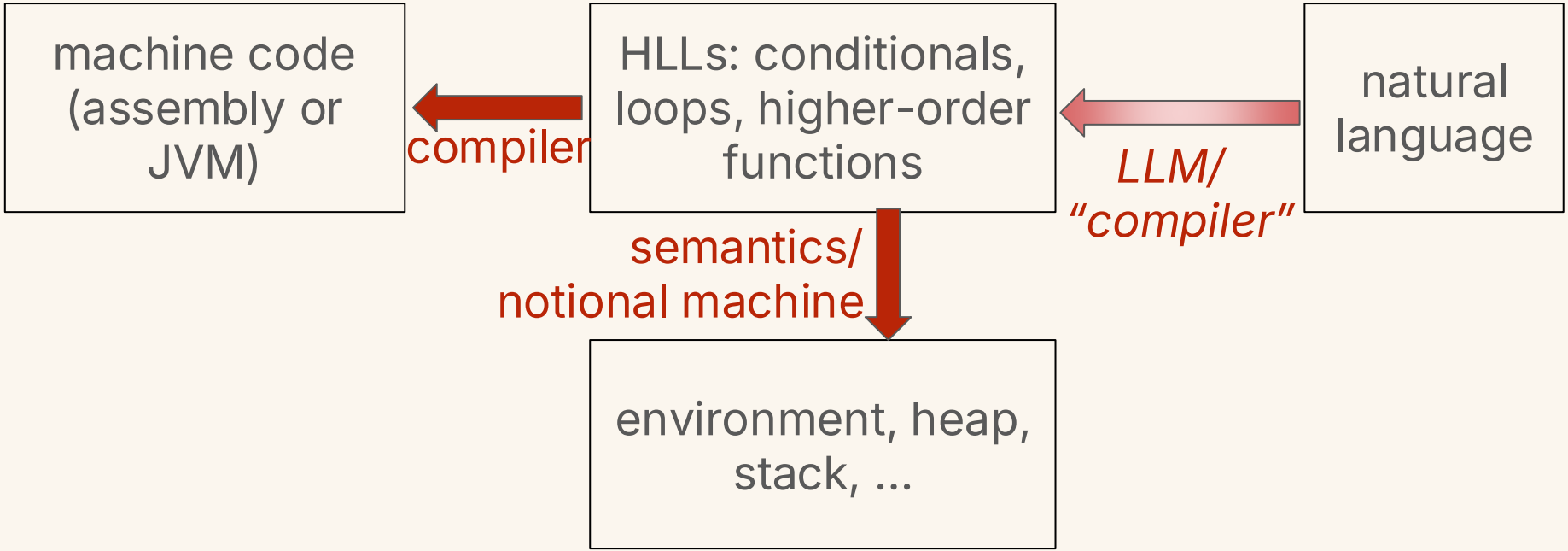
iterate on and peer-critique UI designs



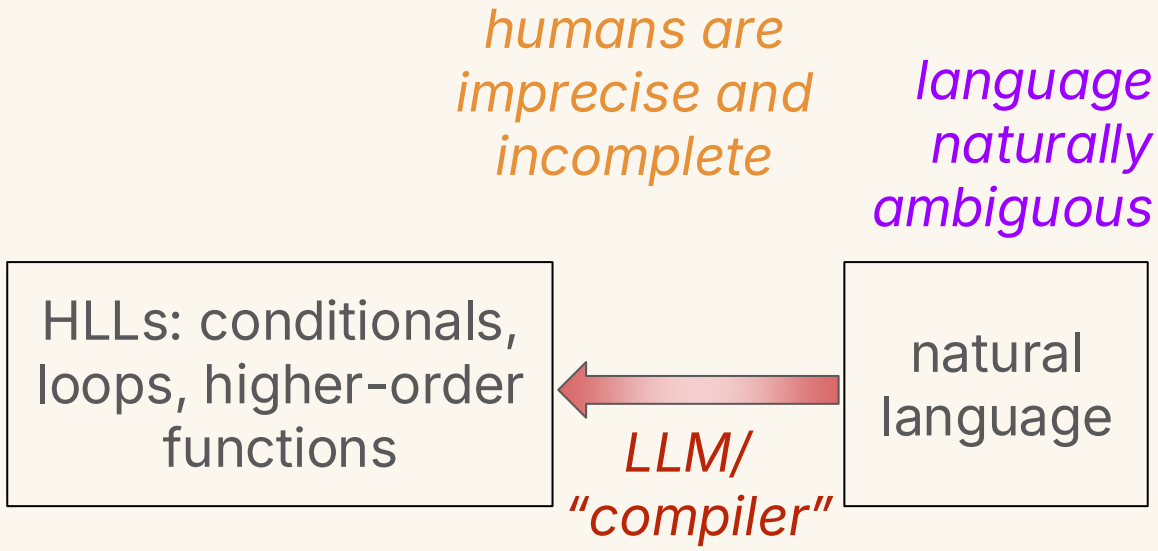
Start easy: coding changes

humans are imprecise and incomplete

language naturally ambiguous



LLMs serve as a flaky compiler



- they fill in missing assumptions
- they make up additional assumptions
- they select answers that resolve imprecision

→ humans should gauge whether results are acceptable

Dealing with Flaky Compilers

This isn’t all that new – human collaborators make mistakes too!

The entire field of software engineering is about building robust software in the presence of miscommunication, imprecision, unimagined situations, failed components, and so on

Revised Learning Objectives

Some software engineering must appear in the first-year CS curriculum



Two Experimental Courses

Agentic Studio, Spring 2026

19 first and second-year students with at least one semester of programming experience

RQ: what kinds of SE methods can early-stage students learn to enable building interesting projects with agents, while still being able to explain and defend the quality of their work

Joint with Michael Littman

Data-Centric Intro with Agents, Fall 2026

figure out what this looks like without programming experience

Experimental Design of Agentic Studio

We identified a **collection of software qualities** to target: testability, maintainability, usability, separation of concerns, MVC

We chose **increasingly-challenging projects** that exposed the qualities

Students kept **reflection journals** throughout, noting how they divided work between themselves and Claude, documenting experience (excitement, frustration, etc), and logging Claude's errors

Standard: justify why someone should pay you for your product

Students worked with $20/month Claude accounts

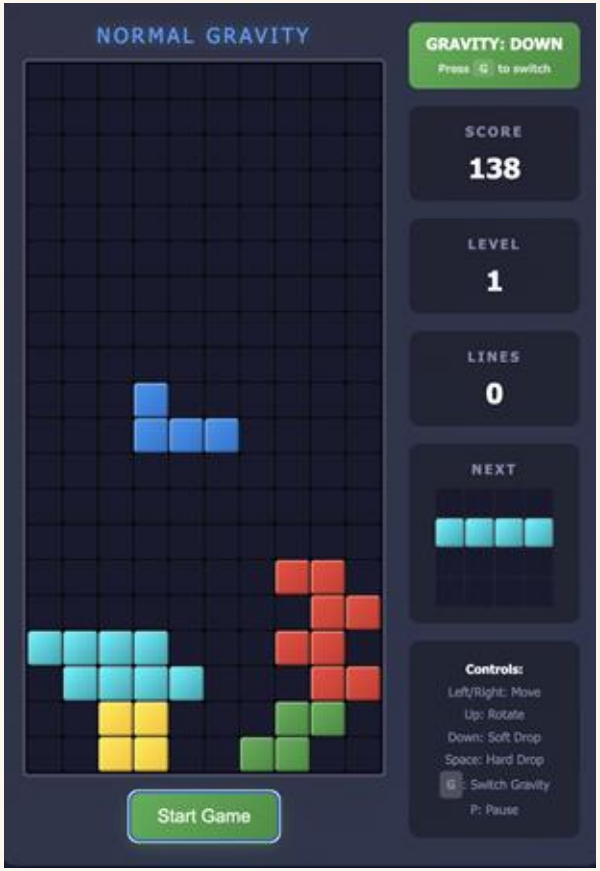
Project Sequence (many subparts per project)

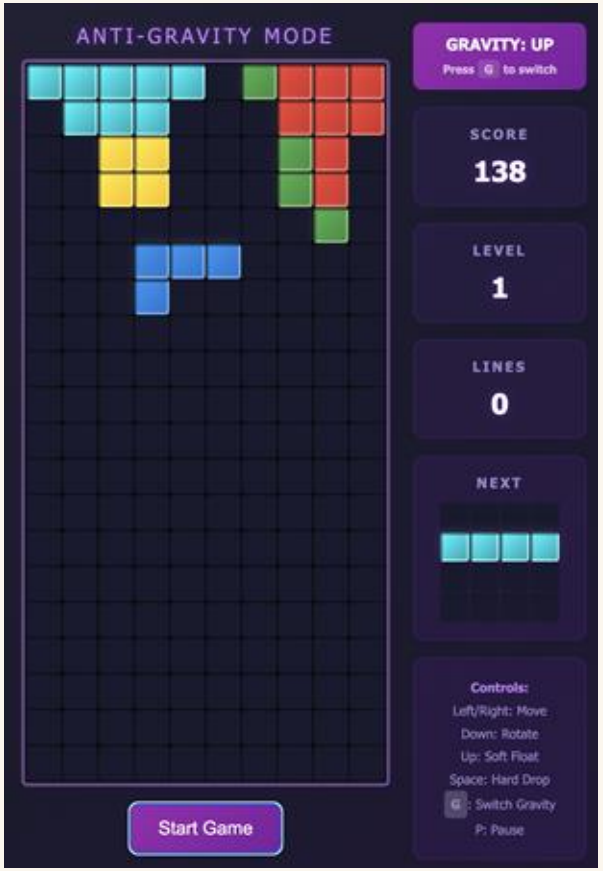
Tetris	classic game, but with reversible gravity
Airport Weather	join two large (messy) datasets for analysis
Requirements Checker	check whether set of courses meets degree requirements with multiple options
Music Tour Planner	plan datatypes/propose data structures for a band's travel schedule
Chat Client	build chat client, calling out to an LLM for summarization and personalization features

> Tetris

> Add a G option that switches block movement between falling and rising

Motivate the need for
- specifications
- MVC
- testing





Requirements Checker

Gave students the webpage with the official Brown CS requirements and the document where we track exceptions, FAQs, etc.

Produce a checker that takes list of courses and determines whether they satisfy the requirements

- various buckets with different options (e.g. 4 intro sequences)
- limits on how many non-technical courses
- same course could satisfy multiple requirements

Requirements Checker

Stage 1 (manual, no AI)

- design a **data representation** that can capture requirements for CS and one other department (of student's choice)
- design a **testing strategy** (specific scenarios, not low-level tests)

In class: **small-group critique** plus class-wide design review of three different approaches

Requirements Checker - Subsequent Stages

- Stage 2 (agentic)
  - implement the checker (in-class review of diff approaches)
  - got a mix of (mostly) backtracking greedy algs and some constraint solving. Few claimed to understand how code worked
- Stage 3 (**peer review**)
- We taught about constraint solving
- Stage 4 (revision)
- **Test Fest!** Everyone's solutions against everyone's tests (incl. mine)

Peer Review Rubric (Readability Example)

Types are used to add clarity

Below Expectations	**Meets Baseline Expectations**	**Strongly Meets Expectations**
Types are either missing or generic (like “list” or “dict” or “any”).	Type names reflect problem-specific kinds of data (like “gradebook”, “studentInfo”).	Types are used consistently on parameters and function outputs. Searching the code on a type name helps reveal how values of that type flow through the system.

Peer Review Rubric (Maintainability Example)

Configuration that will come from end-users can be done in formats that make sense for those users

Below Expectations	**Meets Baseline Expectations**	**Strongly Meets Expectations**
Configuration is hard-coded into the program.	Configuration is done through an external file with a reasonable format, and the code has reasonable well-formedness checks.	Configuration is done through one or more external files with reasonable formats, and the program does all the relevant checking that the configuration is well-formed and valid.

Peer Review Rubric (Testability Example)

Code is designed to be tested

Below Expectations	**Meets Baseline Expectations**	**Strongly Meets Expectations**
Large portions of the functionality are in monolithic functions that make it hard to test the intermediate computations.	Major steps of the computation are in helper functions that can be tested independently, but there are still portions of internal code that should be moved into functions for separate testing.	All key operations on the data model are isolated in helper functions that can be tested directly.

Finding: Software Engineering Part Went Well

Students learned to write good design documents and specifications

Did well writing test strategies, variation in how many actual tests

Journals revealed four trust relationships with Claude:

- Plan carefully, validate plan, then trust Claude
- Don't trust it, work in small chunks to validate output
- Never fully trust, not confident they can verify
- Work with Claude to plan, skim plans, then trust Claude

did well in code reviews

only 1-2 students, stuck in "productivity" mindset

Finding: Students Want to Understand High Level

Chat Client assignment was pretty bumpy

- Students didn't understand client/server going in
  - we gave one in-class lecture at start of project
  - otherwise told students to self-study with Claude (didn't work)
- Once Claude's messages were beyond what they could understand, students reported disengaging and letting Claude do more of the work
- Didn't help that instructor initially misconfigured the servers

Overall

Both the students and the instructors learned a lot

- students reported using skills in internships and later courses
- instructors confident that agents have a valuable role in motivating and supporting software engineering skills in first year
- the accountability message resonated, but so did doing projects where students felt ownership (a more internal measure)

But what if the students don’t already know how to program?

Is this (still) feasible?
Is this even responsible?

ONCE WE GIVE THEM AI-AGENTS, THEY'LL NEVER DO THEIR OWN WORK AGAIN!

Students are going to use agents whether we allow it or not

People can now do both real harm and neat things with little to no programming experience.
What's our responsibility?

We should provide a counter-narrative to vibe coding

We know a lot about creating reliable code!

111E – Data-centric Intro with Agents

Intro course that combines manual programming, writing code plans, designing datatypes, and agentic coding

- higher-order functional programming over tables and lists
- developing data organizations and semi-formal specifications
- testing, including property-based testing
- articulating constraints on data
  - and how constraints differ from other kinds of errors
- LLMs only at first (Pyret+Gemini gems), then Claude+Python

Example: Budget-Conscious Shopping (month in)

Given a shopping list with items and quantities and pricing data from three stores, determine (a) which store gives the lowest price on each item and (b) which store would yield the lowest total cost.

```
def cheapest_store_for_item(item):
    """Return (store, price) for the lowest unit price of item, or (None, None) if no store sells it."""
    best_store = None
    best_price = None
    for store in store_prices:
        if item in store_prices[store]:
            price = store_prices[store][item]
            if best_price is None or price < best_price:
                best_store = store
                best_price = price
    return best_store, best_price

def build_annotated_list():
    """Return (plan, unavailable). plan maps store -> list of
       (item, quantity, unit_price, cost). unavailable lists items no store sells."""
    plan = {}
    for store in store_prices:
        plan[store] = []
    unavailable = []
    for item in shopping_list:
        quantity = shopping_list[item]
        store, price = cheapest_store_for_item(item)
        if store is None:
            unavailable.append(item)
        else:
            plan[store].append((item, quantity, price, quantity * price))
    return plan, unavailable
```

what does a student need to make sense of this (Claude) code?

3 onions
1 coffee
2 chickpeas

item	#
onion	3
coffee	1
chickpeas	2

add columns that pull from store data

item	#	mega	corner	market
onion	3	.70	.80	.75
coffee	1	4.29	3.89	4.59
chickpeas	2	1.59	1.89	1.79

compute new column

item	#	mega	corner	market	store
onion	3	.70	.80	.75	mega
coffee	1	4.29	3.89	4.59	corner
chickpeas	2	1.59	1.89	1.79	mega

select cols and order rows

item	#	store
coffee	1	corner
onion	3	mega
chickpeas	2	mega

pivot/ summarize

store	total
mega	9.57
corner	10.07
market	10.42

add columns that pull from store data

item	#	mega	corner	market
onion	3	.70	.80	.75
coffee	1	4.29	3.89	4.59
chickpeas	2	1.59	1.89	1.79

compute new column

item	#	mega	corner	market	store
onion	3	.70	.80	.75	mega
coffee	1	4.29	3.89	4.59	corner
chickpeas	2	1.59	1.89	1.79	mega

select cols and order rows

item	#	store
coffee	1	corner
onion	3	mega
chickpeas	2	mega

```
fun lookup-price(item :: String, prices :: Table) -> Number:
  matches = filter-with(table, lam(row): row["item"] == item end)
  matches.row-n(0)["price"]
end

t1 = build-column(shopping-list, "mega",
  lam(r): lookup-price(r["item"], mega-data) end)

t2 = build-column(t1, "corner",
  lam(r): lookup-price(r["item"], corner-data) end)

t3 = build-column(t2, "market",
  lam(r): lookup-price(r["item"], market-data) end)

fun lowest-price(mega :: Number, corner :: Number, market ::
Number) -> Number:
  best = num-min(mega, corner, market)
  if best == mega: "mega"
  else if best == corner: "corner"
  else if best == market: "market"
  end
end


t4 = build-column(t3,
  lam(r): lowest-price(r["mega"], r["corner"], r["market"]) end)

order-by(t4.select-columns([list: "item", "quantity", "store"]),
   "store")
```

```
def cheapest_store_for_item(item):
    """Return (store, price) for the lowest unit
       price of item"""
    best_store = None
    best_price = None
    for store in store_prices:
        if item in store_prices[store]:
            price = store_prices[store][item]
            if best_price is None or price < best_price:
                best
                best
    return best_stor

def build_annotated_
    """Return (plan,
       -> list of (i
       unavailable l
    plan = {}
    for store in stor
        plan[store] = []
    unavailable = []
    for item in shopping_list:
        quantity = shopping_list[item]
        store, price = cheapest_store_for_item(item)
        if store is None:
            unavailable.append(item)
        else:
            i = (item, quantity, price, quantity * price)
            plan[store].append(i)
    return plan, unavailable
```

Aligning abstractions for planning and programming supports data-rich programs and may help students with auditing code

```
fun lookup-price(item :: String, prices :: Table) -> Number:
  matches = filter-with(table, lam(row): row["item"] == item end)
  matches.row-n(0)["price"]
end

t1 = build-column(shopping-list, "mega",
  lam(r): lookup-price(r["item"], mega-data) end)

                                          "corner",
                                          (r["item"], corner-data) end)

                                          'market",
                                          (r["item"], market-data) end)

                                          :: Number, corner :: Number, market ::

                                           corner, market)
                                          mega"
  else if best == corner: "corner"
  else if best == market: "market"
  end
end


t4 = build-column(t3,
  lam(r): lowest-price(r["mega"], r["corner"], r["market"]) end)

order-by(t4.select-columns([list: "item", "quantity", "store"]),
   "store")
```

Example: Bikeshare Analysis

Given data on where bikes were in the morning and the rides taken that day, propose 15-20 overnight moves of bikes (by a truck) to support the next day's operations.

what could go wrong?

- data might not be clean or normalized — constraints
- rides might not be properly tabulated into end-of-day totals — specs
- chosen metric might not be scored properly — testing

Adding a simple dashboard opens discussions about UI

Findings so far

We haven’t done bikeshare yet, but have done shopping tables

Students are doing just fine with

- tables programming (all manual, checked by live code review)
- describing constraints on data
- describing plans precisely (they later iterate with Gemini)

Students still working on

- differentiating constraints from errors
- designing good tests outside of errors and constraint violations
- learning to talk about code in weekly code reviews

Stepping Back

Vision: first-year CS **can and should** cover basics of building reliable applications

Skills:

- produce code, both manually and with agents
- create code plans
- identify problem constraints
- produce validators
- create testing strategies
- audit code and artifacts for testability, maintainability, usability, and match to specs

Knowledge:

- common data structures
- constraint solvers
- SQL
- limits of greedy algorithms
- relationships between data structures and runtime/space
- protecting data and privacy
- considering social impacts

Many new assignment opportunities

- Write a spec, have agent give three different interpretations or implementations, then argue for a particular one
- Propose two different data organizations for a problem, have agent develop sufficiently large test data to try the two ideas experimentally
- Given a piece of agent-generated code, have conversation with the agent about how the spec has been satisfied (or not)
- Staff use agent to generate sizeable pool of broken solutions, see how many student test suite detect

What goes in first vs second semester?

manual coding, (mostly) higher-order functions

natural language with LLM/agents producing code



semi-formal specs; tests and validators in code

weaving SE with manual coding

~~machine code (assembly or JVM)~~

moved later in the curriculum



~~HLLs: conditionals, loops, higher-order functions, objects~~

should this still be in CS1?

CS2?

natural language (with specs, design docs, and testing plans)

new CS1?



Much more research is needed!

There's already research showing that **students can learn to develop good test suites** in first-year CS

There's also (now somewhat old) research on the **challenges that people have writing formal specifications**

How does this manifest with semi-formal specifications for product spaces that students understand well as end-users?

Much more research is needed!

There's already research showing that **people don't necessarily retain feedback from AI interactions**

There's also substantial cognitive science research on using **contrast and variation to design activities for learning and retention**

How do we design learning activities where students use agents for feedback on (software engineering) tasks and actually absorb the lessons from that feedback?

Closing with Grace

There are both students and faculty who have principled objections to AI and its use (these are varied and multifaceted)

Humans are more concerned about what they might lose than what they might gain

- Prospect Theory, Kahneman and Tversky

The most damaging phrase in the language is “We’ve always done it this way!

Grace Hopper, Info Week, 1987

I had a running compiler, and nobody would touch it because, they carefully told me, computers could only do arithmetic; they could not do programs