Observations on the Design of Program Planning Notations for Students

Elijah Rivera
elijah_rivera@brown.edu
Brown University
Providence, RI, USA

Shriram Krishnamurthi
shriram@brown.edu
Brown University
Providence, RI, USA

Kathi Fisler
kfisler@brown.edu
Brown University
Providence, RI, USA

ABSTRACT

Program planning is the process of splitting a problem description into subtasks that can be solved independently, then composed into a solution. While much has been written about planning since the 1980s, little research looks at modern contexts such as programs to process data tables. Tool support for this sort of planning is even rarer. As part of a project to develop such tools, we have run two studies to try to identify steps, representations, and interactions that would support novice university students in planning and programming multi-task programs that process data tables. This experience report describes our observations so far, while also raising questions about how to make planning useful for students.

CCS CONCEPTS

• **Social and professional topics → Computing education**.

KEYWORDS

Program planning, program decomposition, plans

ACM Reference Format:
Elijah Rivera, Shriram Krishnamurthi, and Kathi Fisler. 2024. Observations on the Design of Program Planning Notations for Students. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2024), March 20–23, 2024, Portland, OR, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3626252.3630901

1 INTRODUCTION

Programmers frequently decompose problem descriptions into smaller pieces, which they solve and then combine into a complete solution. *Planning* refers to strategic approaches to this process in which at least some aspects of the decomposition and composition activities are outlined prior to implementing the final code. Planning is arguably a critical component to programming and system design, especially for multi-person projects.

Novice programmers can also leverage planning when figuring out how to get started on problems and projects that extend beyond what they have written before. In an ideal world, planning could make the programming process more manageable for many students: it could help them outline solutions and provide a structure in which to try and revise approaches to problem tasks before they get mired in the low-level details of specific implementations.

This sounds good in theory, and has for over 40 years (see section 2). Yet planning is rarely taught, and tools to help students with it—and especially to provide them feedback while planning—hardly exist. The community has not yet arrived at consensus for tools and representations for creating and assessing students' plans or approaches to planning.

We have been trying to teach planning explicitly to novice university students, including creating tools for the process. In this, we tried building on prior work [7, 14] that had success with students who had some prior programming experience. In contrast, we have *not succeeded* at tool designs that work well for our novice students. This experience report describes the things we've tried and summarizes our observations from each. We hope that others who are trying to teach planning might benefit from our experience.

2 RELATED WORK

Early work on planning focused on how students construct code, contrasting situations where students were solving problems with familiar structures versus new ones. Spohrer and Soloway's model of plan composition in Marcel [15] suggested that novices start from an initial plan then refine it as needed to add tasks and handle errors. Rist's model of planning for new problems posited that students first identify a recognizable *focal computation* in a program, then build out the rest of the program around that [12, 13]. Castro [1] also found evidence of Rist's focal computations among novice students who were developing novel programs via functional programming.

Soloway and Spohrer observed that composing plan fragments appears to be harder for novices than either decomposing problems into subtasks or producing code for familiar subtasks [15]. This work was done in the context of imperative programming with loops and I/O, in which code for different subtasks would need to be interwoven at the level of individual lines, rather than run sequentially or composed through function calls. In our studies, students use higher-order functions and built-in table operations on problems for which sequential and conditional composition suffice.

Only a couple of projects have studied pedagogic strategies for teaching planning. Muller et al. developed "pattern-oriented instruction" in which students learned a collection of named patterns and were taught to label and identify these patterns in solving new problems [11]. de Raadt et al. gave students a "strategy guide" for integrating plans [3]. Both of these efforts centered around imperative programming and their idiomatic constructs.

Cunningham et al. developed *purpose-driven programming*, an approach to planning, programming, and learning code structures in domain-specific contexts [2]. In their curriculum, learners work with domain-specific plans for common tasks (e.g., deleting files or searching for all matching tags in a website). Developing a program starts with decomposition into goals and subgoals phrased in terms of the domain-specific tasks. Code schemas for those subgoals are retrieved from a library, then tailored to the problem details. Their proposed plan-and-code development tool starts with code comments to label the goals and subgoals (building on work on subgoal labeling in CS education [9, 10]). Our work is more general purpose (it does not fix a domain), though the names of the higher-order and built-in functions with which our students process tables do carry more semantic information than general-purpose loops.

Representations of plans and how they evolve (notationally) during planning has also been the subject of recent research. Our prior work proposed representing plans as compositions of higher-order functions [14]. That student population (in an accelerated-pace intro course, most with some programming in high school) was effective at recognizing which higher-order functions applied to problem statements and at using these functions as a skeleton for plans [7]. This work also explored the idea of adapting Snap*!* [6] to planning via custom blocks. Our paper takes inspiration from this work, instead trying similar ideas with novice university-level programmers and for programs that process tables rather than lists.

Scholars have argued that planning is only suitable for some people. Levi-Strauss' concept of bricolage [8], which involves a more experimental approach to designing solutions, has been applied to computing education [16]. Planning assumes certain cognitive skills, which might not apply to all in neurodiverse populations. We do not claim that planning will work (the same way) for all learners; knowing how different students decompose and/or compose tasks and code, however, is still important for programming instruction.

3 THE ALLURE OF PLANNING

Figure 1 provides a problem that illustrates what our department expects students to be able to solve sometime during their first semester of learning to program. In a nutshell, students are given two CSV tables—one with prices of art pieces in some currency and the other with currency conversion rates—and asked to produce the price of an art piece in a different currency than its baseline one. Samples of the tables and desired computation appear in fig. 1.

Figure 2 outlines the key steps in the computation (these align with the table-manipulation operations presented in our lectures). Students could organize the final code in different ways, depending on which helper functions they create, where they introduce variables, and whether they abstract otherwise duplicated computations into helper functions. At the time of this assignment, students have been shown only one way to extract a row from a table (using a built-in higher-order function to find rows based on predicates).

Having assigned this or similar problems for many years, certain challenges arise frequently. Some students can't get started (decomposition challenges). Some do, but try to process both tables simultaneously (subtask-separation challenges). Some get code that works but the final structure of their helper functions or expressions is unwieldy (composition challenges). We hypothesize (or at least want to believe) that support for planning would help. We are trying to teach students to extract and follow an outline at the level of granularity in fig. 2 (though not necessarily in that format).

Problem statement: Companies that do business internationally need to be able to quote prices in different currencies, following a table of exchange rates. Your starter code defines two tables: one specifies the baseline currency and price of art pieces by their ID numbers; another specifies the conversion rate (the multiplicative factor) to convert from one currency (from-c) to another (to-c).

Submit a plan for a function called *get-art-in-1* that computes the price of a piece of art in a specific currency. The function takes in an art table, a currency table, the id of an art piece, and a payment currency; it returns the price of the art item in the desired currency based on the data in the two tables. For this problem, assume that the currency table has a row with the base currency in the first column and the desired currency in the second column.

For example, given the following two tables, we expect *get-art-in-1* to report that the price of art piece 24 in EUR would be $120 * 0.05$.

Art price table

id	cost	currency
23	25	USD
85	100	CHF
24	120	MXN
59	500	JPY
56	55	USD
87	72	JPY

Currency conversion table

from-c	to-c	rate
USD	EUR	0.99
EUR	CHF	1.05
MXN	EUR	0.05
JPY	USD	0.0068
USD	MXN	20.00

Figure 1: The currency-conversion problem

(1) Find the row of the art-price table with the given art ID
(2) Extract the cost and baseline currency for the piece from that row
(3) Find a row in the currency-conversion table with the baseline currency in the `from-c` column and the target currency in the `to-c` column
(4) Extract the conversion rate from that row
(5) multiply the art piece cost by the conversion rate

Figure 2: A currency-conversion solution outline

4 WHAT MAKES FOR USEFUL PLANNING?

We view *planning* as a process by which a problem statement is decomposed into parts that can be implemented, then composed into a solution. A *plan* outlines the decomposed parts and how they will fit back together. Planning could be viewed as a process in which a plan is created before coding begins, and program elements are developed and composed according to the plan. Such a clean view is unrealistic in practice. Part of planning involves understanding the problem, which can involve writing some pieces of code. This could lead to refinement of the plan.

What, then, makes for *useful* planning? Regardless of whether a plan is fully created prior to coding or is developed concurrently with bits of coding, we posit that useful planning helps a student work on subtasks of a problem in isolation, while still keeping track of how each subtask fits into a bigger picture. Thus, **the subtasks in a plan should be at a granularity that the student knows how to solve** (say, because they have previously solved similar problems). This implies that *different students could have different plans for the same problem*. For a student to work productively from a plan, **the subtasks (and their intended compositions) should also be readily implementable in code**. Since different languages and platforms have different constructs at different levels of abstraction, this implies that *plans are not necessarily language-agnostic*. This may be contentious, especially since plans should allow for multiple implementations. With the end goal of planning and plans being the production of code, however, we must assume that students will be thinking about what pieces they can build, and how, as part of the planning process; language features affect this, especially for novices who know only a handful of constructs.

Our challenge, then, is to figure out how to teach, support, and assess planning in ways that help students

- make sense of problems,
- track high-level subtasks,
- sketch out how subtasks compose, and
- adapt when initial ideas fail to work as conceived.

Each of these pieces raise their own questions about how to represent that piece and how to support a programmer in working productively between the pieces and the eventual code.

5 OUR TEACHING CONTEXT AND STUDENTS

We work at a highly selective university in the USA. We ran the experiments in this paper across two consecutive offerings of a CS (computer science) course for novices (here called CS-NOV). The course follows the "data-centric" approach [5] to combine data science and computer science concepts. Students learn to write programs to process and analyze data tables (loaded from CSV or Google Sheets), as well as basic data structures (lists, records, trees, dictionaries). The course starts in functional programming, then transitions to Python for stateful programming and working with Pandas. The course attracts roughly 200 students per semester from all class years and many intended majors. Most students do not intend to major in CS, but rather want to learn some programming and computing to apply to work in other disciplines.

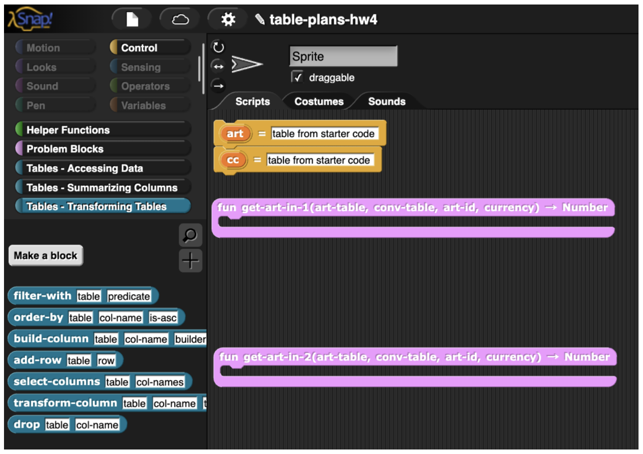
6 OUR EXPERIENCES

Recall that our focus is on experiments that inform building tools to help students with planning. Here, we look at two approaches. The first was tool-driven and based on promising prior work, but effectively failed. The second set aside tooling to focus on notations and was more successful.

6.1 Planning with Blocks

Inspired by previous work showing that university students with prior programming experience could effectively construct viable plans using higher-order functions [14], we decided to see whether this also applied with the novice students in CS-NOV. The previous work had created custom Snap*!* [6] blocks to enable drag-and-drop planning; we did the same in our context of table operations.

Figure 3 shows a screenshot of the setup that we provided our students for the currency-conversion problem (fig. 1) during Fall semester 2022. Students were given a palette of blocks corresponding to operations on tables that they had been learning (see lower



```
art = table from starter code
cc = table from starter code
fun get-art-in-1(art-table, conv-table, art-id, currency) → Number
fun get-art-in-2(art-table, conv-table, art-id, currency) → Number
filter-with table predicate
order-by table col-name is-asc
build-column table col-name builder
add-row table row
select-columns table col-names
transform-column table col-name
drop table col-name
```

Figure 3: Blocks for planning table functions (the lower block for *get-art-in-2* was for a related problem that this paper is not discussing)

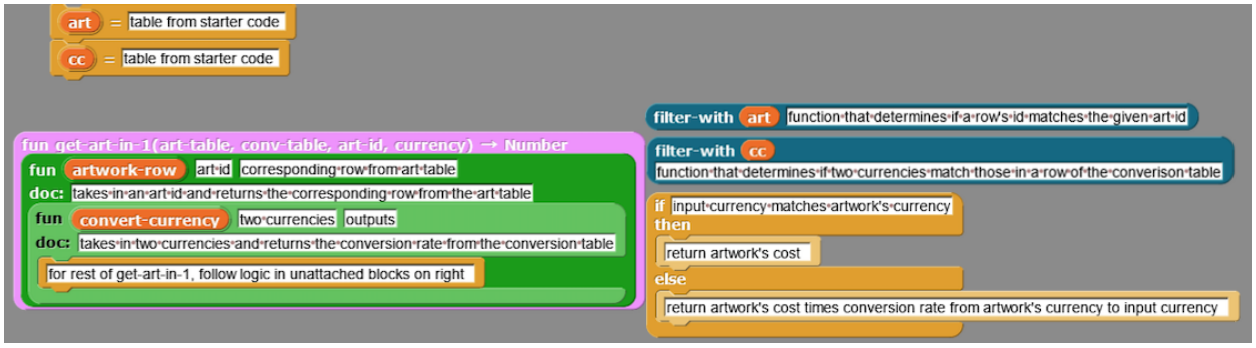
left of fig. 3). Unlike with conventional blocks programming, however, these table-operation blocks allowed both nested blocks and, crucially, *free-form text* within the holes in each block.

The ability to use free-form text enables planning as opposed to coding: a student can describe what should happen within that block without writing any code. Each student could individually choose to use prose at the point where they felt a textual description would suffice, and could control the level of detail in that prose. Nesting of blocks provides a structured notation for subtask composition.

As instructors, we liked that Snap*!* programs could be saved (and submitted) as XML files: we hoped to create analysis tools that read the XML and provided students with automated partial feedback on the quality of their plans (e.g., did it follow one of the expected shapes). Completely free-form text or diagrams would be hard to parse to give feedback. We believe that students are more likely to use program design tools that give actionable feedback, so this feature was important to our instructional goals.

Usage: Students used the blocks-based planning tool in three assignment contexts: they were introduced to it during a TA-facilitated (ungraded) lab (roughly a month into the course), then they used it on the assignment containing the currency-conversion problem (one week later), then they optionally used it as part of a course project with a large design component (two weeks later).

Experience: Students used the blocks in different ways on the currency-conversion problem. Figure 4 shows a submission that took advantage of free-form text in blocks. This solution sets up a couple of helper functions (*fun*) blocks, but doesn't try to nest all of the blocks (though the prose indicates the intended nesting). The function arguments needed to filter the tables (the *filter-with* blocks in the right column) are described only in prose. In contrast, the solution in fig. 5 puts complete code into the free-response boxes (along with small bits of prose). Some of those code fragments define new functions; some use conditionals. The solution does not use the built-in blocks for these constructs. (The free-form areas are not code-editor windows, so the code in those areas lacks indentation.)

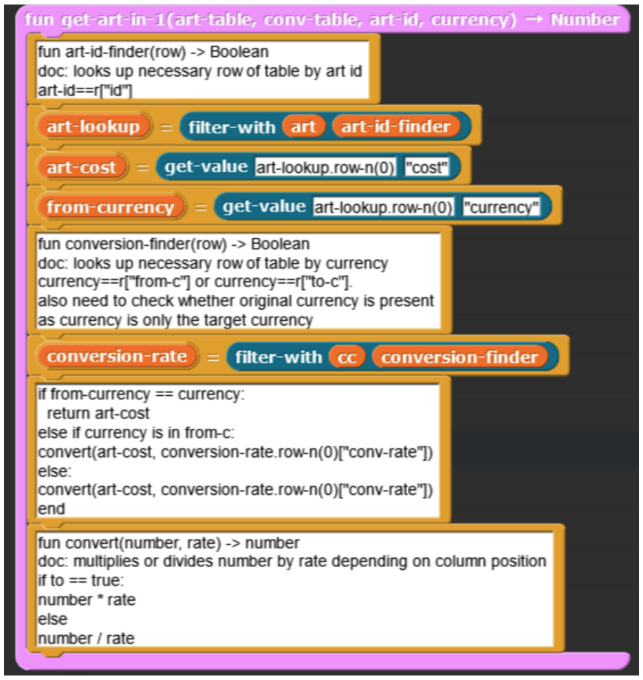


```
art = table from starter code
cc = table from starter code

fun get-art-in-1(art-table, conv-table, art-id, currency) → Number
  fun artwork-row art-id corresponding row from art table
  doc: takes in an art id and returns the corresponding row from the art table
    fun convert-currency two currencies outputs
    doc: takes in two currencies and returns the conversion rate from the conversion table
      for rest of get-art-in-1, follow logic in unattached blocks on right

filter-with art function that determines if a row's id matches the given art id
filter-with cc
function that determines if two currencies match those in a row of the converison table
if input currency matches artwork's currency
then
  return artwork's cost
else
  return artwork's cost times conversion rate from artwork's currency to input currency
```

Figure 4: A student plan with text in blocks



```
fun get-art-in-1(art-table, conv-table, art-id, currency) → Number
  fun art-id-finder(row) -> Boolean
  doc: looks up necessary row of table by art id
  art-id==r["id"]
  art-lookup = filter-with art art-id-finder
  art-cost = get-value art-lookup.row-n(0) "cost"
  from-currency = get-value art-lookup.row-n(0) "currency"
  fun conversion-finder(row) -> Boolean
  doc: looks up necessary row of table by currency
  currency==r["from-c"] or currency==r["to-c"].
  also need to check whether original currency is present
  as currency is only the target currency
  conversion-rate = filter-with cc conversion-finder
  if from-currency == currency:
    return art-cost
  else if currency is in from-c:
  convert(art-cost, conversion-rate.row-n(0)["conv-rate"])
  else:
  convert(art-cost, conversion-rate.row-n(0)["conv-rate"])
  end
  fun convert(number, rate) -> number
  doc: multiplies or divides number by rate depending on column position
  if to == true:
  number * rate
  else
  number / rate
```

Figure 5: A student plan with code in blocks

We conjecture that the student for fig. 5 may have written much of the code first, then gone back to create plans (we know many students did this based on discussion board posts and office hours). However, the lowermost box (with the definition of *convert*) does not contain valid code: there are errors with syntax, unbound variables, and the table contents. This student may then have interleaved planning and coding. Regardless, this "plan" is much closer to code than we were hoping to see. The plan in fig. 4 was much closer to what we expected and hoped for.

During the assignment, many students expressed uncertainty regarding the level of detail and style that the instructor was looking for in plans. Students did not feel they had been given sufficient instruction regarding expectations (despite the lab, a lecture, and a separate course resource on planning); a few explicitly asked how to maximize their grades. For at least some students, planning was seen as a requirement rather than an actual aid to their programming process. Other students, in contrast, came to office hours with plans in hand and sought help in working between plans and code.

The Problem of Conditionals. Conditionals, in particular, created confusion regarding the boundaries between plans and code. We had emphasized in lecture that plans were meant to be high-level outlines of how a computation would be done; they should not be as detailed as final code. None of the examples that we covered in lecture or in lab needed (or used) conditionals. The currency-conversion problem doesn't need them either. The only conditional behavior is in the table *filter-with* operation, which takes a *predicate* (as a function) as an input. For the final currency-conversion code, this predicate simply returns the result of an equality test between two values (as shown in the top free-form box in fig. 5). However, many students in CS-NOV found it confusing to program directly with Boolean expressions, and instead used them in an *if* (a style we noticed in their programming that seemingly carried over to their plans as well). Indeed, we more often saw *if*-expressions at the code level as in fig. 5 than at the planning level as in fig. 4.

Feedback. Overall, however, students had extensive complaints about doing block-based planning. These were sufficiently numerous that we made the use of blocks optional on the course project, even though students had to produce *some* form of plan. Out of 104 project submissions, only four students appeared to use Snap*!*. Instead, most students wrote to-do list style outlines (like fig. 2).

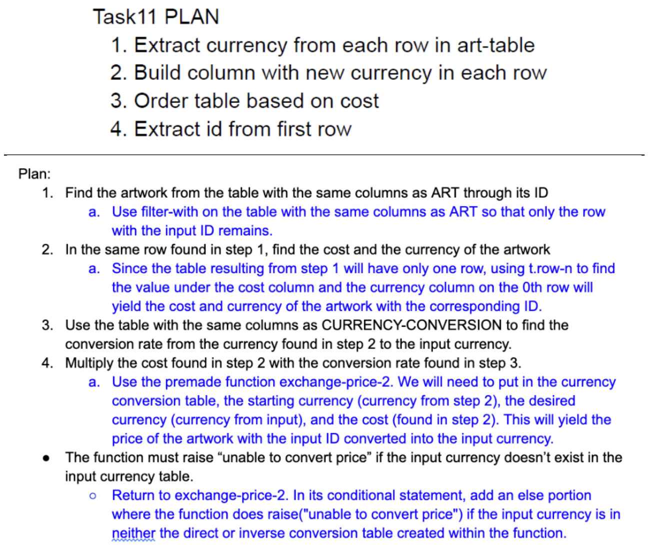
To better understand what happened, we added an optional ungraded question to the final exam asking students about their experience with the blocks-based planner. Out of 195 students who took the final, 138 responded. Representative comments include:

> *Snap! was time-consuming & unhelpful b/c they required too many parts that were not realistic to my thought process when actually coding.*

> *I planned all my code in writing and after writing the code did Snap!.*

> *I plan using pen and paper and Snap! is too structured.*

> *Snap! felt like busywork to me mostly. I mean it was helpful to some degree for cementing how different code*


Task11 PLAN

1. Extract currency from each row in art-table
2. Build column with new currency in each row
3. Order table based on cost
4. Extract id from first row

Plan:

1. Find the artwork from the table with the same columns as ART through its ID
    a. Use filter-with on the table with the same columns as ART so that only the row with the input ID remains.
2. In the same row found in step 1, find the cost and the currency of the artwork
    a. Since the table resulting from step 1 will have only one row, using t.row-n to find the value under the cost column and the currency column on the 0th row will yield the cost and currency of the artwork with the corresponding ID.
3. Use the table with the same columns as CURRENCY-CONVERSION to find the conversion rate from the currency found in step 2 to the input currency.
4. Multiply the cost found in step 2 with the conversion rate found in step 3.
    a. Use the premade function exchange-price-2. We will need to put in the currency conversion table, the starting currency (currency from step 2), the desired currency (currency from input), and the cost (found in step 2). This will yield the price of the artwork with the input ID converted into the input currency.

- The function must raise "unable to convert price" if the input currency doesn't exist in the input currency table.
    - Return to exchange-price-2. In its conditional statement, add an else portion where the function does raise("unable to convert price") if the input currency is in neither the direct or inverse conversion table created within the function.

Figure 6: Two to-do list plans at different detail levels

pieces fit together: conditionals, helper functions, etc., but I think it took more time than it's worth.

My thought process isn't like snapping LEGOs together [...] I do understand teaching how to plan is important [but let] students upload a picture/PDF of their written/typed checklist so there's flexibility on how to think (arrows, bullet points, etc) and then very leniently grade and/or analyze those.

After processing the student work and feedback, we concluded that the blocks-based planning experiment had failed in CS-NOV (in contrast to the earlier findings [14] with more experienced students; we return to this contrast in section 7). We decided to return to the drawing board, have students plan in more free-form style in the next semester, and see what insights we might gain about providing tool support for planning for novice CS students.

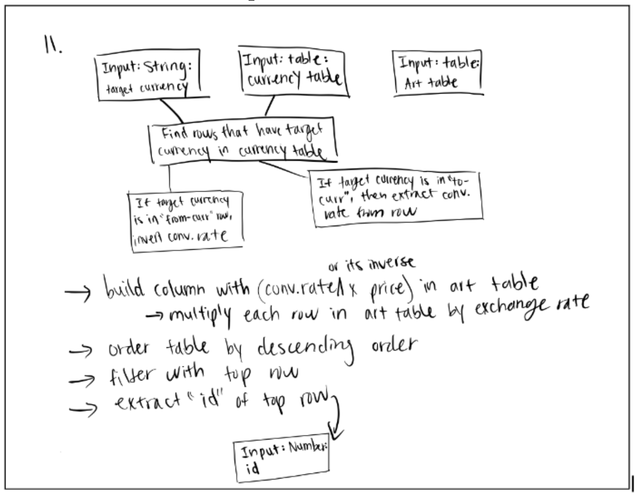
6.2 Open-Ended Planning of Table Functions

Given that students perceived blocks as too rigid (despite the ability to write free-form text inside the holes), in Spring 2023, we relaxed the medium and let students use plain paper. We also gave them complete freedom in how they wanted to present their plans. For guidance, we showed students both a textual form of plan like a to-do list (similar to fig. 2), and a visual one using diagrams. Students saw plans for 3-4 programs across lecture and lab (with more examples of the textual form). We informed students that they were free to use any format that appealed to them, even if it did not match the above two. Crucially, we did *not* show the block-based notation or offer them Snap*!*. We asked students to experiment and comment on their choices. Students did so by turning in this information and their plan (typically scanned to PDF) alongside their final code.

Results. We again consider the currency-conversion problem. The resulting plans were mostly "correct" in their correspondence to the problem statement, but at different levels of detail. Figure 6 shows examples of both minimal and detailed plans. The minimal plan is not entirely correct (the third step for ordering based on cost is not appropriate for the problem). The detailed plan makes creative use of indentation and color to outline implementation plans for each high-level step (we had not shown something like this in lecture or lab). Nearly all of the to-do list plans were at one of these extremes (minimal or with code or implementation detail).

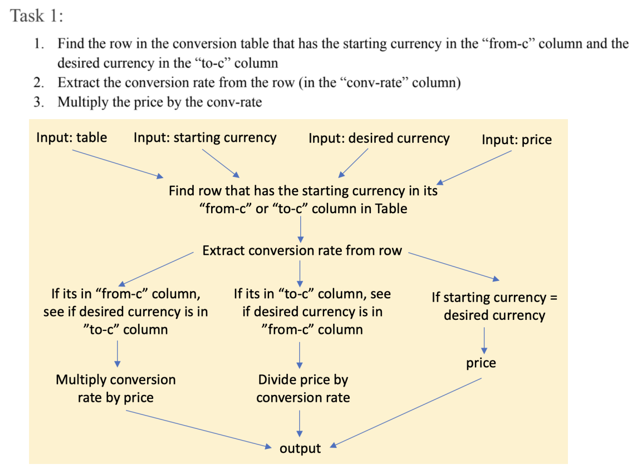
Mixing Notations. Most students used just one notation in their plan; the vast majority were to-do lists. We were surprised that only six students used more than one, especially since it would be easy to do so on paper. We invited each of the six for a private in-person interview about their choices. We explained that we were trying to design tools to support planning, so we were also interested in their thoughts on how tooling could help their planning process. This section discusses the feedback from three students. Each was working on an expanded version of currency conversion in which the two currencies could appear in either order in the table.

Student A. This student used a both a diagram and a to-do list. The diagram covered the extraction of the conversion rate from the currency table, while the to-do list covered the use of that rate to determine the converted price from the second table.


11.
Input: String: target currency
Input: table: currency table
Input: table: Art table
Find rows that have target currency in currency table
If target currency is in "from-curr" row, invert conv. rate
If target currency is in "to-curr", then extract conv. rate from row
→ build column with (conv.rate or its inverse x price) in art table
→ multiply each row in art table by exchange rate
→ order table by descending order
→ filter with top row
→ extract "id" of top row
Input: Number: id

The student reported that planning was useful because the operations they wanted to use manually didn't line up with those in the programming language. Planning with the diagram felt "dumber", but planning with text seemed too similar to writing code, which created a mental block for the student. Regarding choosing between the two, the student said "When I could visualize in my mind, I only needed text. When it doesn't fit in my head, I made a diagram". The student reported that they tried going back to the plan when they got stuck, but usually found that their mistakes were in the low-level code rather than the plan. Overall, the student wished for "frameworks for approaching problems" as they were used to in social science research or the scientific method (CS-NOV already teaches the design recipe from *How to Design Programs* [4]; our planning work is an attempt to expand on it).

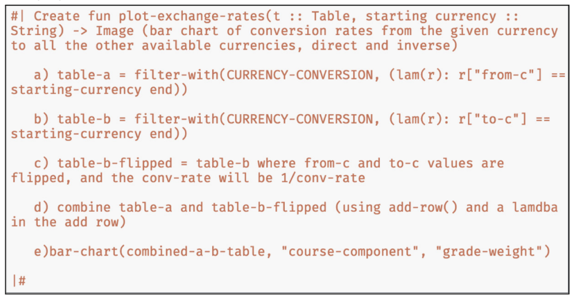
Student B. This student tried both a high-level text to-do list plan and a more detailed diagram-based plan. They found that text and the ability to scribble down ideas informally was easier. The student reported putting more detail into the plan for "fear that we were going to get counted off". They switched from planning to coding when planning grew tiresome. They also mentioned that which device they were using affected their plan representations: "maybe it would've been easier to make diagrams on my laptop, but I like working on my iPad". When asked whether they thought tool support for planning would be useful, the student replied "I tend to use what I'm familiar with, so even if there's a tool I probably wouldn't learn it". In general, they were more interested in having more instructions or examples rather than a tool.


Task 1:

1. Find the row in the conversion table that has the starting currency in the "from-c" column and the desired currency in the "to-c" column
2. Extract the conversion rate from the row (in the "conv-rate" column)
3. Multiply the price by the conv-rate

Input: table
Input: starting currency
Input: desired currency
Input: price
Find row that has the starting currency in its "from-c" or "to-c" column in Table
Extract conversion rate from row
If its in "from-c" column, see if desired currency is in "to-c" column
If its in "to-c" column, see if desired currency is in "from-c" column
If starting currency = desired currency
Multiply conversion rate by price
Divide price by conversion rate
price
output

Student C. This student wrote plans as comments in their code. They reported trying to engage with planning, but found they already think in code and "Didn't like having two artifacts out of sync". The student preferred a code-based format to free-form text: "I don't like doing it in all English, because I find it very difficult to look and determine what's what. I like using variable names for that." This student did find planning helpful, both for seeing the decomposition and because plans give a way to collect signatures for functions that would eventually be needed, but having a planning tool was not attractive: "Naming variables as steps, possibly with English headers, would be my most productive version of the plan because then I could just go straight into the code".


```
#| Create fun plot-exchange-rates(t :: Table, starting currency ::
String) -> Image (bar chart of conversion rates from the given currency
to all the other available currencies, direct and inverse)

   a) table-a = filter-with(CURRENCY-CONVERSION, (lam(r): r["from-c"] ==
starting-currency end))

   b) table-b = filter-with(CURRENCY-CONVERSION, (lam(r): r["to-c"] ==
starting-currency end))

   c) table-b-flipped = table-b where from-c and to-c values are
flipped, and the conv-rate will be 1/conv-rate

   d) combine table-a and table-b-flipped (using add-row() and a lamdba
in the add row)

   e)bar-chart(combined-a-b-table, "course-component", "grade-weight")

|#
```

7 DISCUSSION AND FUTURE WORK

Our experiences across the two semesters raised interesting trade-offs between block-based and free-form notations for planning. In general, we saw more variation in the nature of plans with the block-based tool. The "free form" plans were nearly all to-do lists with similar high-level steps, though some steps were more detailed than others. With blocks, students focused more on control flow using either if-expressions or named functions and function calls; the vast majority of plans were either function-oriented, conditional-oriented, or to-do list style sequences of steps. The provided blocks shaped the plan structures. In contrast, the lack of structure in "free form" mode seemed to result in to-do lists being a default.

In contrast to our previous data [14], the CS-NOV block-based plans seemed messier and more code-like. We later realized that the problems in the previous paper had required boolean predicates, but not conditional logic. Students who used to-do lists sometimes captured conditionals via nested bullets; others wrote separate bullets for if and else without nesting. Those working in blocks could use explicit if-blocks or if-like expressions in free-form text. Since there was no way in Snap*!* to nest blocks within free-form text, however, once students wrote if-expressions, they stayed in code-like notation. This raises future work questions about how to represent conditional branches in plans.

Proportionally more students in CS-NOV expressed frustration with Snap*!* than in the previous study. Some students found it confusing to learn a second notation; many complained that the blocks didn't offer (enough) benefit. Perhaps students already understood how to do the problems, weren't sure what good plans look like, or didn't know how to leverage plans to write code. This needs further study, as students will (reasonably) resist planning if they do not perceive benefits to doing so. Explicitly teaching how to use plans when coding and debugging might also help convey benefits.

Returning to our "useful planning" criteria from section 4, this paper highlights the need for deeper study of how notations support planning for composition, which was where student plans most often fell short. Little research has looked explicitly at this, despite existing work showing that it causes students more difficulty than decomposition [1, 15]. Some to-do list plans leave the mechanisms for composing steps unspecified, leaving students stuck after they implement individual steps. Some students put explicit variable names in plan steps to support later composition, but most do not. Diagram-based plans use arrows or lines to connote all of sequential control-flow, conditional control-flow, and data flow; this overloading of arrows could cause later confusion. Future studies of planning should focus on forms of expressing composition, how they might help students structure code, and how to encourage students to think about planning beyond decomposition.

Studying composition would also enrich conversations about structured planning versus bricolage in CS education. Bricoleurs may prefer experimentation to structured decomposition, but both approaches eventually end up having to compose parts of solutions into effective wholes. How do students who take each approach deal with composition, and how might we better support them at doing so? We still have much to learn from studying planning across its own component subtasks.

ACKNOWLEDGMENTS

Thanks to CS-NOV staff and students for their participation, and to Jens Mönig for help configuring Snap*!*. The first author thanks his mom for helpful discussions about teaching and learning. Work partly supported by NSF award SHF-2227863.

REFERENCES

[1] Francisco Enrique Vicente Castro and Kathi Fisler. 2016. On the Interplay Between Bottom-Up and Datatype-Driven Program Design. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (Memphis, Tennessee, USA) *(SIGCSE '16)*. ACM, New York, NY, USA, 205–210. https://doi.org/10.1145/2839509.2844574

[2] Kathryn Cunningham, Barbara J. Ericson, Rahul Agrawal Bejarano, and Mark Guzdial. 2021. Avoiding the Turing Tarpit: Learning Conversational Programming by Starting from Code's Purpose. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 61, 15 pages. https://doi.org/10.1145/3411764.3445571

[3] Michael de Raadt, Richard Watson, and Mark Toleman. 2009. Teaching and Assessing Programming Strategies Explicitly. In *Proceedings of the Eleventh Australasian Conference on Computing Education - Volume 95* (Wellington, New Zealand) *(ACE '09)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 45–54. http://dl.acm.org/citation.cfm?id=1862712.1862723

[4] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to Design Programs: An Introduction to Programming and Computing* (2 ed.). MIT Press. https://htdp.org/

[5] Kathi Fisler, Shriram Krishnamurthi, Benjamin S. Lerner, and Joe Gibbs Politz. 2023. *A Data-Centric Introduction to Computing*. Published on-line. https://dcic-world.org/, accessed 2023-07-29.

[6] Brian Harvey and Jens Mönig. 2010. Bringing "No Ceiling" to Scratch: Can One Language Serve Kids and Computer Scientists?. In *Proceedings of Constructionism 2010: Constructionist Approaches to Creative Learning, Thinking and Education: Lessons for the 21st Century*. Library and Publishing Centre, Facutly of Mathematics, Physics and Informatics, Comenius University, Bratislava, 1–10.

[7] Shriram Krishnamurthi and Kathi Fisler. 2021. Developing Behavioral Concepts of Higher-Order Functions. In *ACM Conference on International Computing Education Research*.

[8] C. Levi-Strauss. 1962. *The Savage Mind*. University Of Chicago Press.

[9] L.E. Margulieux, B.B. Morrison, and A. Decker. 2020. Reducing withdrawal and failure rates in introductory programming with subgoal labeled worked examples. *International Journal of STEM Education* 7, 19 (2020). https://doi.org/10.1186/s40594-020-00222-7

[10] Lauren E. Margulieux, Mark Guzdial, and Richard Catrambone. 2012. Subgoal-Labeled Instructional Material Improves Performance and Transfer in Learning to Develop Mobile Applications. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research* (Auckland, New Zealand) *(ICER '12)*. Association for Computing Machinery, New York, NY, USA, 71–78. https://doi.org/10.1145/2361276.2361291

[11] O. Muller, B. Haberman, and D. Ginat. 2007. Pattern-oriented instruction and its influence on problem decomposition and solution construction. In *Proceedings of ITiCSE*. ACM, New York, NY, 151–155.

[12] Robert S. Rist. 1989. Schema Creation in Programming. *Cognitive Science* (1989), 389–414.

[13] Robert S. Rist. 1991. Knowledge Creation and Retrieval in Program Design: A Comparison of Novice and Intermediate Student Programmers. *Hum.-Comput. Interact.* 6, 1 (Mar 1991), 1–46.

[14] Elijah Rivera, Shriram Krishnamurthi, and Robert Goldstone. 2022. Plan Composition Using Higher-Order Functions. In *ACM Conference on International Computing Education Research*. https://doi.org/10.1145/3501385.3543965

[15] James C. Spohrer and Elliot Soloway. 1989. Simulating Student Programmers. In *International Joint Conference on Artificial Intelligence*. 543–549.

[16] Evelyn Stiller. 2009. Teaching Programming Using Bricolage. In *Proceedings of the Consortium for Computing Sciences in Colleges*. 35–42.