Substance Beats Style: Why Beginning Students Fail to Code with LLMs

Francesca Lucchetti
Northeastern University

Zixuan Wu
Wellesley College

Arjun Guha
Northeastern University

Molly Q Feldman
Oberlin College

Carolyn Jane Anderson
Wellesley College

Abstract

Although LLMs are increasing the productivity of professional programmers, existing work shows that beginners struggle to prompt LLMs to solve text-to-code tasks (Nguyen et al., 2024; Prather et al., 2024; Mordechai et al., 2024). Why is this the case? This paper explores two competing hypotheses about the cause of student-LLM miscommunication: (1) students simply lack the technical vocabulary needed to write good prompts, and (2) students do not understand the extent of information that LLMs need to solve code generation tasks. We study (1) with a causal intervention experiment on technical vocabulary and (2) by analyzing graphs that abstract how students edit prompts and the different failures that they encounter. We find that substance beats style: a poor grasp of technical vocabulary is merely correlated with prompt failure; that the information content of prompts predicts success; that students get stuck making trivial edits; and more. Our findings have implications for the use of LLMs in programming education, and for efforts to make computing more accessible with LLMs.

1 Introduction

There is a growing body of evidence that large language models (LLMs) are increasing the productivity of professional programmers (Etsenake and Nagappan, 2024). At the same time, previous work shows that students struggle to leverage LLMs in programming across a variety of tasks and models (Nguyen et al., 2024; Prather et al., 2024; Mordechai et al., 2024). But why is this the case?

Prior work has reported on students' and instructors' perception of why student-LLM interactions go wrong, positing many explanations including unfamiliarity with technical vocabulary (Nguyen et al., 2024; Feldman and Anderson, 2024; Mordechai et al., 2024; Prather et al., 2024), model non-determinism (Lau and Guo, 2023; Vadaparty et al., 2024), and trouble understanding LLM output (Nguyen et al., 2024; Vadaparty et al., 2024). However, there is little quantitative evidence about these potential sources of miscommunication.

In this paper, we test two competing hypotheses about the cause of student-LLM miscommunication. One possibility is that students provide all of the information that the model needs, but use language that models cannot understand. Non-expert programmers talk about code differently than experts, leading to problems for models trained largely on expert code. A second possibility is that students do not understand what information a model needs to solve a given problem. Writing prompts involves decisions about what information the model may be able to infer from pretraining versus what information must be stated directly in the prompt. These decisions may be more challenging for students to make, since they do not yet have a strong sense of what information code typically contains.

This paper tests the impact of these potential error sources in two sets of experiments on a dataset of 1,749 prompts authored by 80 students (Babe et al., 2024). To isolate the effect of linguistic variation, we conduct a causal analysis of lexical choices for technical terminology by replacing them with near-synonyms used by students. To study information selection, we annotate series of prompts in student problem-solving attempts with problem-specific "clues," or information that describes the intended behavior of generated code.

Overall, our findings reveal that student-LLM coding difficulties spring from challenges in selecting relevant information rather than challenges with technical vocabulary. Our study of the information content of prompts shows that prompts with missing clues almost always fail. Moreover, students typically get "stuck" in cycles because they make trivial edits to prompts instead of changing their information content. Our causal analysis of prompt wording finds relatively weak effects of

Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), pages 8541–8610

modifying technical terminology. Although certain substitutions can hurt prompt success rates, correcting non-standard terminology rarely improves them. This suggests that the relationship between technical vocabulary and prompt success is more correlational than causal.

Taken together, our results provide empirical evidence that the information content of student prompts is what matters, rather than their (mis)use of technical vocabulary. These findings have strong implications for the use of LLMs in programming education and, more broadly, for efforts to broaden the accessibility of computing with LLMs.[1]

2 Related Work

As the use of LLMs for programming has become widespread, the question of prompt wording has become increasingly important. Early work revealed high sensitivity to prompt wording on programs (White et al., 2023; Döderlein et al., 2023), which has efficiency implications (Mozannar et al., 2024). Several techniques that address prompt wording (Strobelt et al., 2022; Oppenlaender, 2023; Zamfirescu-Pereira et al., 2023; Ma et al., 2024). Liu et al. (2023) take a user-centered approach to teaching strategies for prompting. Döderlein et al. (2023) study keyword removal and replacement. Zhu-Tian et al. (2024) generate program sketches from Python keywords in prompts. Xia et al. (2024) automatically reword existing task descriptions for more robust code generation benchmarks.

Novice Programmers and LLMs. LLMs have have sparked much discussion in computing education (Finnie-Ansley et al., 2022). There is a growing body of work studying how students use LLMs in computing classes (Zamfirescu-Pereira et al., 2023; Prather et al., 2023; Kazemitabaar et al., 2023a; Denny et al., 2023; Mordechai et al., 2024; Vadaparty et al., 2024). A convergent finding is that students struggle to leverage LLMs. Many potential explanations have been advanced: Lau and Guo (2023)'s study of CS educators discusses model non-determinism as a barrier; Prather et al. (2024) explores the cognitive load imposed by code suggestions; and Kazemitabaar et al. (2023b) describe over-reliance on the model. Finally, multiple studies posit that technical language is a barrier between students and LLMs (Nguyen et al., 2024; Feldman and Anderson, 2024; Mordechai et al., 2024; Prather et al., 2024).

This paper uses the dataset by Babe et al. (2024), which contains 1,749 prompts from students who have completed one college programming course. Babe et al. (2024) turn their dataset into a benchmark to measure LLM performance on novice-written prompts. They report some correlations between technical terms and prompt success. Nguyen et al. (2024) study student experiences during the experiment, including students' self-perceptions of why the task is challenging: they highlight prompt wording as a key student-perceived barrier. This is reaffirmed in Feldman and Anderson (2024)'s replication with students with no coding experience.

Prompting Effects in Generative Models. There is a large set of existing work exploring the effect of different prompting techniques for LLMs more broadly. Prior work has shown that models are surprisingly robust to misleading, corrupted, or irrelevant prompts (Webson and Pavlick, 2022; Min et al., 2022; Madaan et al., 2023; Ye and Durrett, 2022; Khashabi et al., 2022; Wang et al., 2023). In this light, the documented issues that novice programmers experience when working with LLMs for programming are surprising. Our work may help to reconcile these two bodies of work by exploring the cause of student-LLM miscommunications.

Terminology in Other Generative Domains. The impact of prompt terminology has been studied in non-code domains. For text generation, previous work has studied prompting techniques to control style (Yeh et al., 2024; Raheja et al., 2023). Text-to-image models are very sensitive to choices in keywords (Liu and Chilton, 2022), limiting their usability for some applications (Tseng et al., 2024) and users (Chang et al., 2024).

3 Dataset

Our goal is to understand what it is about student-written prompts that makes them less effective for LLM code generation. We use the StudentEval dataset released by Babe et al. (2024), who use a subset of their data to benchmark LLMs for code generation. Unlike many datasets of programming prompts, it contains many different prompts per task, including multiple submissions by the same author, allowing us to explore both wording choices and how the information content of

[1]We make the code and data for our experiments available at https://github.com/nuprl/substance-vs-style.

Function signature

```
def total_bill(grocery_list, sales_tax):
```

Tests

```
total_bill([['eggs', 6, 0.99], ['milk', 1, 1.49], ['bread', 2, 3.5]], 0.07) # 15.44
total_bill([['eggs', 6, 0.99], ['milk', 1, 1.49], ['bread', 2, 3.50]], 0.0) # 14.43
total_bill([['bread', 2, 3.50]], 0.5) # 10.5
```

Docstring Attempt 1 (generated code fails some tests)

you will have two inputs a list of lists and the tax rate. for every list in the list of lists multiply the second and third item and add all of them and then multiply that by the sales tax plus 1

Docstring Attempt 2 (generated code passes all tests)

you will have two inputs a list of lists and the tax rate. for every list in the list of lists multiply the second and third item and add all of them and then multiply that by the sales tax plus 1. if the resulting number has more than two decimal places shorten it to two decimal places.

Figure 1: An example problem that a student solves in two attempts. Given the function signature and tests, they write the first docstring. The platform prompts the model to generate the function body from the function signature and docstring (not the tests), and then tests the generated code. From the failed tests, the student realizes that the model needs to be told to round to two decimal places. They add this clue in the second prompt, which succeeds.

prompts is edited during a prompting session.[2]

The dataset contains 1,749 prompts written by 80 students who had completed exactly one programming course. They were asked to complete problems drawn from a set of 48 CS1 programming tasks exercising a range of programming concepts. The dataset was collected in a prompting experiment that worked as follows (Figure 1): (1) the student was shown 3-5 test cases and asked to write a Python docstring for the function; (2) the experimental platform prompted an LLM (*code-davinci-002*) to generate a Python function, conditioned on the function signature and the student-written docstring; (3) the experimental platform tested the generated function on the provided tests; and (4) the student could try again or give up and move on to the next problem. Each student did 8 problems.

We use different subsets of the StudentEval dataset to explore our research questions. To study the effect of information content on prompt success, we consider problems where at least five students submitted multiple times (33 tasks). We exclude tasks that were trivial (all students succeeded at first try) or that were attempted by few students, since these are uninformative. To study the effect of prompt wording, we select a lexically diverse subset by taking each student's first and last prompts per problem (953 prompts). We study only the first and last prompts because there is often little lexical variation in intermediate prompts (as shown in our information content experiment), which would skew the results of our lexical analysis.

[2]The dataset contains sequences of prompt-edits, but their benchmark uses only the first/last prompt by each student.

4 Methods

Our work explores the impact of two potential causes of student-LLM miscommunication: how students word their prompts, and how students select information to include in their prompts.

4.1 Measuring the Impact of Prompt Wording

To understand how students' wording of prompts affects model performance, we use a counterfactual causal inference approach. We systematically measure the impact of wording related to what Mordechai et al. (2024) refer to as the "structured language" that experts use "to describe the logical control flows within the desired program." We define a set of key programming concepts and systematically substitute alternative terms used by students to measure how the success of their prompt would have been impacted by alternative wording.

4.1.1 Tagging Concept References

We select 12 key technical concepts that occur frequently in the StudentEval dataset, including references to data types (e.g., list, string, dictionary), operations on data (e.g., concatenate, append, typecast), and terms related to data flow and control flow (e.g., input, loop, return).

For each concept, two expert annotators identified every lexical variation used to refer to these concepts in the prompts. The tag set includes tags for all morphological variants of a given lemma, to ensure that the substitutions match the capitalization and tense of the original terms. In addition, three sets of tags were used for terms referring to function input, to capture different syntactic structures. The full tag set contains 78 tags for 14 category lemmas. See Appendix E for the annotation

Original: *Convert the input into integers and check if it is a prime number.*
Tagged: *$Typecast:Convert$ the $parameter:input$ into $integers:integers$ and check if it is a prime number.*
Substitution: *Convert the input into whole numbers and check if it is a prime number.*

Figure 2: An example of tagging and then substituting "integer" with "whole number".

procedure and all lemmas.

Overall, references to these concepts appear 4,262 times across the dataset. Collapsing variations of the same lemma within a prompt (e.g., "string","strings"), we find that the median number of technical terms per prompt is three and the maximum is ten. Figure 2 shows an example of how three concept references in a prompt get tagged.

4.1.2 Replacement Sets

We identify the most common terms that students use to refer to to each concept category. An initial list was developed by reading through all prompts in the Nguyen et al. (2024) and Feldman and Anderson (2024) datasets, to get the widest possible set of variations. We computed frequencies for terms in this initial list and selected terms used at least twice in StudentEval. This led to a final set of 65 substitution terms, with at least two substitutions for each of the 14 concept lemmas.

4.1.3 Causal Analysis

We conducted term-by-term substitution experiments across 65 category-replacement pairs. For each category-replacement pair, we replaced all expressions tagged with the category using the replacement lemma. Terms tagged with other categories were left unchanged, with the category tags removed and the original terms restored.

Figure 2 shows an example of the term-by-term substitution on a tagged prompt, where we replace all terms tagged with category *integer* with the replacement term *whole number*. Our tagging retains information about the tenses, plurals, and capitalization of the original words. In this example, *integers* tagged with *integers* is replaced with *whole numbers*. Terms *Convert* and *input* tagged with other categories are unchanged by the substitution.

Using Gpt-4o mini (OpenAI, 2024), Llama 3.1 8B and 70B (Llama Team, 2024), we generate completions for all prompts before and after substitution. A completion is considered correct if it passes all tests for the problem. We compute a pass rate per problem by sampling 200 completions using common hyperparameters for code generation.[3]

4.1.4 Significance Testing

We measure the statistical reliability of observed differences in pass rates using mixed-effects binary logistic regression models that include random effects for prompt ID and problem. The outcome variable is the pass@1 rate.

4.2 Measuring the Impact of Information Content

Another possible source of error is the information content of student prompts. Like other forms of communication, prompting involves a trade-off between communicative efficiency and likelihood of success. An effective prompter seeks to obtain correct results from the LLM while minimizing their own descriptive effort.

A key part of effective prompting, therefore, is understanding the level of detail that is necessary to guide the model. An expert prompter may be able to quickly describe a task in a concise prompt. Novices, on the other hand, may struggle to distinguish cases that need to be specified (e.g., both branches of a conditional) from cases that pattern together, or atypical coding patterns from typical ones. This may be the case even when students fully understand the programming task, since efficient prompt-writing involves guessing what information models can infer without explicit direction.

We seek to understand how the information content of prompts changes over the course of a prompt trajectory. When a prompt fails, are students able to identify what information is missing? Prior work shows that students tend to write successively longer prompts (Babe et al., 2024); in this analysis, we seek to understand whether this additional verbiage contains useful information.

4.2.1 Grouping LLM Outputs by Test Results

When a prompt fails to generate correct code, a prompter must decide how to edit their prompt to improve their chances of success. An edit may add information about the intended behavior, remove information that is distracting or wrong, or simply change how the information is described. By studying how and when students edit the information in their prompts, we gain insight into the relationship between information content and prompt success.

[3]Following Chen et al. (2021), we use top-p sampling (0.95) and temperature (0.2) to calculate pass@1.

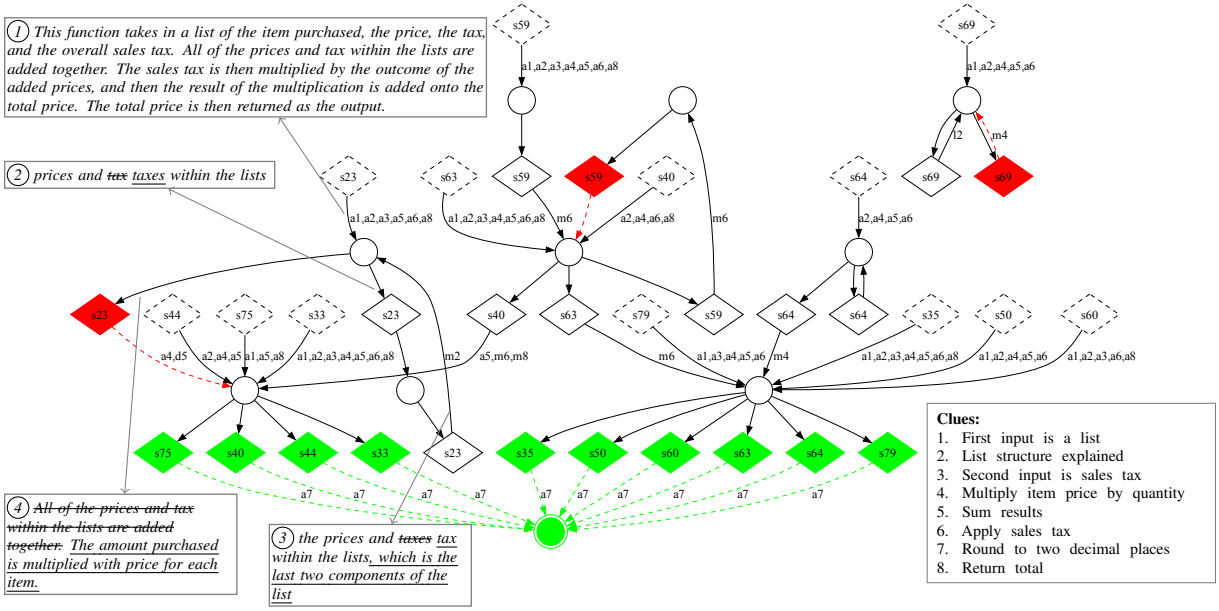

Figure 3: The graph of prompt trajectories for TOTAL_BILL (Figure 1). We highlight the trajectory of S23 who ultimately fails: their first prompts ① has most clues, but omits Clue #7 (bottom right of figure). Their next prompt ② is a trivial change. ③ adds detail about the list structure (Clue #2), but it was already described well so they cycle back to a previous state. Finally, ④ adds the missing Clue #4 (and deletes Clue #5, but it isn't necessary to solve the problem). Here they give up and fail, but many others succeed from this state after adding Clue #7.

To do this, we study a set of 290 *prompt trajectories*: sequences of prompts entered by a student for a particular task, starting from their first prompt and ending with a final prompt that may or may not succeed on the task.

Although prompts vary significantly in wording, we can group them based on their effect: when used to prompt a model, what is the behavior of the generated code? Every problem has a single group of prompts where the tests produce the expected output (successes). In addition, there are multiple states where tests produce incorrect answers or throw exceptions. The ∘-nodes in Figure 3 represent the ten states that students encounter on the TOTAL_BILL problem: the green node is the success state and the others are different failures.

4.2.2 Information in Prompt Edits

We use the notion of a **prompt clue** to study the information content of prompts. A clue is a piece of information about the function's intended behavior. For each problem, we identify a set of clues by examining the information that successful prompts tend to contain, as well as the expert-written prompts from the STUDENTEVAL dataset. We strive for sets of 3-6 clues per problem.

Expert annotators (experienced CS1 educators) developed the set of clues for each problem and used it to annotate each prompt trajectory. We tag the first prompt in each trajectory with the set of clues present. Subsequently, we tag each prompt edit in terms of its information change: adding a clue (a), deleting a clue (d), removing detail from a clue (l), or rewording a clue without removing detail (m). A null tag (0) is used to mark edits that do not change the information content of a prompt.

Figure 3 (bottom right) lists the eight clues for the TOTAL_BILL problem (Figure 1). Some of these clues describe the input and output types (Clues #1, #3, and #8). The remaining clues describe the computation. The edge labels in the graph show how students modify the clues present in their prompts.

4.2.3 Prompt Trajectory Graphs

We define a graph with alternating states of all prompt trajectories for a problem from the sequence of prompts, execution outputs, and expert annotations discussed above. For a given problem, let $s \in S$ be the set of students and $p_{s,i} \in P_{S,\mathbb{N}}$ be the set of prompts indexed by student and attempt number. Let $p_{s,i_{\max}}$ be the final prompt by s. Let EXEC : $P_{S,\mathbb{N}} \to O$ be the mapping from a prompt to its test output, where there is a distinguished

output $o_{\text{OK}} \in O$ where all tests pass.

We construct a directed graph $G = (V, E)$ where $V = O \cup P_{S,\mathbb{N}}$. The graph edges are:

- $\langle p_{s,i}, o \rangle \in E$ where $\text{EXEC}(p_{s,i}) = o$
- $\langle o, p_{s,i+1} \rangle \in E$ if there exists $p'_{s,i} \in P$ and $\langle p'_{s,i}, o \rangle \in E$

A node $p_{s,i_{\max}}$ is a *success node* if $\langle p_{s,i_{\max}}, o_{\text{OK}} \rangle \in E$, and is otherwise a *failure node*. We label edges $\langle p_{s,0}, o \rangle \in E$ with the initial clues for student s. For $\langle p_{s,i-1}, o \rangle, \langle o, p_{s,i} \rangle \in E$, we label the edge $\langle p_{s,i}, o \rangle$ with the edits to the clues made from prompt $p_{s,i-1}$ to $p_{s,i}$.

In Figure 3, the ○-nodes are test result nodes and the ◇-nodes are prompt edit nodes $p_{s,i}$. We label each ◇-node with the student's identifier s.[4] The ◇-nodes with dashed edges represent a student's first prompt and the ◇-nodes colored green or red represent their final prompt (success or failure, respectively). We label edges with clue edits. For convenience, we color each student's last edit edge green (success) or red (failure).

The caption of Figure 3 describes the prompt and clue edits by a student who ultimately fails the task. Other patterns can also be read from the graph. For instance, most students succeed in two attempts after adding a clue about rounding (Clue #7). The three students who never solve the problem get stuck in cycles. The graph also shows a disconnected failure state visited only by student s69, who struggled to describe the input list: the generated code assumes a triply-nested list.

We see the kinds of patterns described above in almost all problems, including longer loops and far more failures in the harder problems. We analyze the structure of these graphs in §6 to understand prompt trajectories in more depth.

5 Results: Style Rarely Matters

We measure the effect of prompt wording through a causal intervention experiment in which we explore a range of lexical substitutions for terms referring to 12 key programming concepts. If what hinders students is their lack of fluency with technical vocabulary, we should be able to improve the pass rate of their prompts by substituting more precise technical vocabulary for their unconventional ways of referring to these concepts. We also measure the effect of word choice on high-quality prompts: by including substitution terms that are commonly used by students but less technically precise, we can test whether they decrease pass rates.

[4]The index i can be inferred, unless the student sees the same output 3+ times.

Lemma	Substitution	8B	70B	GPT
String	character	↓	↓	-
	phrase	↓	-	-
	set of characters	↓	↓	↓
	word	↓	-	-
List	brackets	↓	↓	-
	set of brackets	↓	↓	-
	set	↓	↓	↓
Key	attribute	↓	↓	-
	entry	↓	-	-
	item	↓	-	-
	part	↓	-	-
	variable	↓	-	-
Parameter	argument	-	↑	-
Provide	provide	-	↓	-
Return	display	↓	↓	↓
	print	↓	↓	↓
Loop	go through	↓	-	-
	execute a for loop with	↓	↓	-
	run a for loop through	↓	↓	-
	iterate	-	↓	-
	loop through	-	↓	-
Concatenate	splice	↓	-	-
Skip	remove	↓	-	-
	avoid	↓	-	-
	ignore	-	↓	-
	neglect	-	↓	-
Typecast	cast	↓	-	-
	change	-	↑	-

Table 1: Statistically reliable differences in pass@1 after lexical substitutions: Llama 3.1 8B, Llama 70B and Gpt-4o-mini. ↓ denotes a reliably lower post-substitution pass@1; ↑ denotes a reliable increase; and - indicates no significant difference. All statistically reliable differences involve substituting a standard term with an unconventional term, with $p < 0.05$. Full significance values are reported in Appendix E.4.

5.1 How Much Does Style Matter?

We perform lexical substitutions for the 12 concept categories, comparing the original and post-substitution prompt pass@1 rates using Gpt-4o mini, Llama 3.1 8B and 70B. We test each concept category separately, holding the rest of the prompt constant.

Figure 4 summarizes the results of the 65 lexical substitution experiments. Full model tables with significance values can be found in Appendix E.4. In general, we observe only weak effects of lexical substitution across all categories. For 4 out of 14 concept lemmas, there are no statistically reliable differences between the pass rates for the reworded prompts and the originals (Table 1); moreover, when there are statistically reliable differences, they tend to be small. Contrary to the perceptions of students reported in Nguyen et al. (2024),

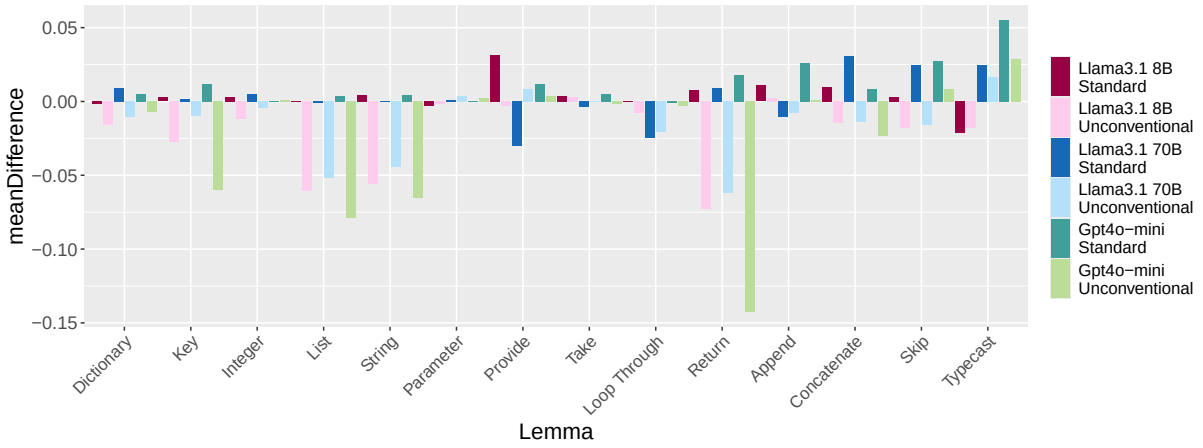

Figure 4: Differences between pass@1 rates before and after lexical substitutions. A negative mean difference represents a decrease in performance after substitution.

technical vocabulary does not seem to have a strong impact on how well models are able to generate code from student prompts.

5.2 Can Rewording Help Failing Prompts?

The overall results show little effect of lexical substitution. Since our substitution sets consist of terms commonly used by students, they include both standard and unconventional ways of referring to the target concepts. For example, students may refer to a string input as "string" (standard) or "word" (unconventional). This means that some substitutions make a prompt less technically precise, while others make it more technically precise.

It is particularly important to understand how prompt wording impacts unsuccessful student prompts. If student word choice is a driving factor in the failure of their prompts, it would be relatively simple to intervene. There are two possible outcomes for low-quality prompts. If the student's vocabulary is causing the low pass rate, then substituting a more precise term should improve its pass rate. On the other hand, the use of unconventional terminology may simply be correlated with poor quality prompts; if this is the case, improving terminology may not lead to higher success rates.

Unlike the analysis in Babe et al. (2024), the lexical substitution experiment enables us to distinguish these two scenarios. We find no evidence of significant gains from fixing unconventional terminology: across all categories, there are no statistically reliable gains from substituting standard terminology (see Appendix E.4).

5.3 When Does Wording Matter?

Our lexical substitution experiments reveal that correcting word choice does not significantly improve pass rates for prompts that use unconventional ways of referring to the target concepts. However, we do observe some statistically significant changes in pass rates: there are reliable negative effects from substituting certain unconventional terms.

We find particularly robust negative effects of diverse unconventional ways of referring to strings: substituting "character" and "set of characters" lower pass rates for string-referring prompts for all models (Table 1). We also find negative effects of unconventional list terms ("brackets", "set", "set of brackets"). The largest magnitude effects are from "set," likely because set is a distinct data type.

For concepts related to control flow, there are interesting differences between input and output concepts. All models are robust to a range of ways of referring to a function's input. However, for return, substituting either "print" and "display" brings pass rates down. This is not surprising: since all of the tasks involve functions that return values, prompting the model to print or display instead is actively misleading. This finding also aligns with the correlational findings of Babe et al. (2024).

Overall, the lexical substitution experiments reveal only weak causal effects of prompt wording. Although substituting unconventional terminology can decrease success rates, correcting unconventional terminology does not seem to help weak prompts. This suggests that the interactions between word choice and prompt success reported in Babe et al. (2024) were correlative, rather than causal: prompts that use unconventional terminol-

ogy are weak for independent reasons.

We view this finding as both surprising, given the body of prior work in which both students and educators identify technical vocabulary as a barrier to working with LLMs, and disappointing, since it would be easier to intervene into student terminology than other aspects of their prompting process.

6 Results: Substance Matters

An alternative hypothesis about student-LLM miscommunication is that students struggle to select the right information for models. We explore this using prompt trajectory graphs (§4.2) for *code-davinci-002* to understand prompt editing. What kinds of edits to information content do students make, and how do they effect the success of their prompts? We focus our discussion on high-level trends; Appendix F contains graphs for each studied task.

6.1 Successful Prompts Have All Clues

We first examine the last prompt in every trajectory. We find that *when all clues for the problem are present in the final prompt, the likelihood of success is 90%*. Conversely, *when even one clue is missing from the final prompt, the likelihood of success falls to 29%*. While these results are drawn from *code-davinci-002* generations, we were able to replicate this finding on current models. For each student's final prompt, we compare the pass@1 score between prompts with all clues versus those with missing clues. We find that Llama3.1 8B achieves a pass@1 score of 50% on prompts with all clues, versus just 14% when there are missing clues; for Llama3.1 70B, the pass@1 scores are respectively 57% and 16%; for Gpt-4o, 86% and 46%. In all models, results indicate that information content is a main factor in the success of student prompts.

There are a few exceptions where students succeed even though their prompts omit clues. We manually inspect these exceptions, which we identify using the prompt trajectory graphs, and find that most fall into one of three cases: (1) the prompt contains hardcoded answers that do not generalize beyond test cases; (2) the function signature has informative names that subsume some clues; or (3) a clue may be technically missing, but duck typing allows the LLM to generate correct code (e.g., the student describes adding strings instead of lists, which uses the same operator in Python).

Considering this, the number of success prompts that are missing one or more clues represents an upper bound on prompt success with partial information. This supports the conclusion that providing all the necessary clues about function behavior is typically what determines prompt success.

6.2 Rewording Existing Clues Hardly Helps

Prompt trajectory graphs illuminate the impact of edits that merely add/remove detail from existing clues, or make trivial edits (edges labelled m, l, or 0 in the graphs). Out of all edges incident to nodes where all tests pass (o_{OK}), we find (1) 28% add detail to an existing clue (m), (2) 11% are trivial rewrites (0) and (3) just 4% remove detail from an existing clue (l). *Rephrasing a prompt without adding a new clue leads to success less than half the time*. Moreover, of these edits, 65% add detail to an existing clue.

Finally, when a prompt contains less than half the clues for a problem, we find that adding/removing detail leads to success only 10% of the time. In other words, the fewer clues a prompt has, the harder it is to succeed by tweaking wording alone. Together, these findings show the impact of information content on prompt success.

6.3 Cycles Involve Uninformative Edits

Prior work shows that students often give up in frustration when their prompt edits do not produce different output (Nguyen et al., 2024). We identify these cycles and measure how hard it is for students to escape them: *when a prompt trajectory has a cycle, its likelihood of eventual success is 30%, compared to 72% without a cycle*. When the cycle exceeds three edges, the likelihood of success drops to 14%. We find a moderate negative correlation between success and cycle length ($r(290) = -0.42$, $\rho =< 0.0001$).

Examining the edits in cycles, we find the majority (90%) involve missing clues. Furthermore, most cycles edits (66%) are exclusively rewrites (l, m, or 0); of these, 43% do not change the level of detail in any clues (0). This shows that students get stuck in a cycle of failing prompts when they are missing important information.

How do students escape? Of the 44 prompt trajectories that manage to break out of a cycle, only 7 have trivial edits. Most escape by adding a new clue (13) or adding detail to existing clues (20). Taken together, our results show that the most successful strategy is adding information, but that most students in cycles simply try trivial wording changes.

Prompt: *This function takes the input of a dictionary. If the key is a planet, it takes the entry and adds it to the total mass. The function outputs the total mass of all planets in the dictionary.*

```
def planets_mass(planets):
  total_mass = 0
  for key in planets:
    if key in planets:
      total_mass += planets[key]["mass"]
  return total_mass
```

Figure 5: Variable/concept confusion.

6.4 When Does Style Matter, Revisited

Overall, our findings support the view that the information content of prompts is more important that wording. However, there are a handful of cases where prompts fail even with all clues.

Figure 5 shows a prompt that succinctly states all clues for the problem. However, the model cannot disambiguate between "planets" as a parameter name and as a general concept, and ends up translating the instruction *if the key is a planet* into `if key in planets`. In other cases, the model interprets language in a surprising way. Three students experienced the same model error in a task to capitalize every other letter in a string: the model produced code that followed their instructions, but also rearranged the string so that all the uppercase letters came first (Figure 15 in the Appendix).

The remaining exceptions can be found in Appendix F.2. Overall, we observe that these failures stem from ambiguity in natural language or model limitations rather than technical vocabulary issues.

7 Conclusion

By investigating two commonly espoused concrete hypotheses about why students struggle to effectively prompt LLMs for code, our work sheds light on what it means for students to write "good prompts." Our results suggest that it is the (lack of) information in prompts, rather than how the information is communicated, that causes student-LLM miscommunication. Although these findings imply that attempts to help student prompters by suggesting alternative wording are unlikely to be very useful, by providing the first empirical evidence of the source of student struggles, we hope our findings will guide future work on teaching prompting towards more impactful interventions.

Limitations

This work builds on the existing StudentEval dataset, which was collected from 80 students in early 2023. These students were selected from three institutions and all had taken only one programming course. Babe et al. (2024) argue that they are representative of beginning students, but they are not representative of students with more programming experience. Our findings may not generalize to more advanced programmers.

The prompts we study were written by students using *code-davinci-002*, which was state-of-the-art at the time, but is now an older model. A newer model, such as a chat model, would lead to different interactions. However, Babe et al. (2024) show that their benchmark remains challenging for several newer models. We re-evaluate StudentEval using Gpt-4o mini, Llama 3.1 8B and 70B and also find that the prompts remain challenging.

The set of categories and terms we explore in our causal inference experiments are specific to the Babe et al. (2024) and Feldman and Anderson (2024) user populations. These students attend select US institutions, therefore their wording choices represent a certain level of English proficiency. The set of substitutions would differ with speakers of other natural languages, as might their effect.

The clues used to tag prompt trajectories represent an expert annotator's perception of the information that successful prompts typically contain. There may be other ways to formulate the same problem. However, we studied all exceptions to our finding and did not find cases where students appeared to use a different set of clues than what the expert annotator found (§6.4).

Ethics Statement

The main ethical concerns surrounding this work lie in its study of student interactions with LLMs. This work uses the public, fully anonymized version of the StudentEval dataset. Therefore, this work has no additional ethical considerations beyond those described in the ethics statement of Babe et al. (2024). The secondary analysis of existing data that we do is consistent with the intended use of the dataset, which is to study how students write prompts.

Acknowledgements

We thank the ARR reviewers for their thoughtful feedback. This work is partially supported by the

U.S. National Science Foundation (SES-2326174 and SES-2326175). This material is based upon work supported by the U.S. Department of Energy, Office of Science under Award Number DE-SC0025613.

References

Hannah Babe, Sydney Nguyen, Yangtian Zi, Arjun Guha, Molly Feldman, and Carolyn Anderson. 2024. StudentEval: A Benchmark of Student-Written Prompts for Large Language Models of Code. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 8452–8474, Bangkok, Thailand and virtual meeting. Association for Computational Linguistics.

Douglas Bates, Martin Mächler, Ben Bolker, and Steve Walker. 2015. Fitting linear mixed-effects models using lme4. *Journal of Statistical Software*, 67(1):1–48.

Ruei-Che Chang, Yuxuan Liu, Lotus Zhang, and Anhong Guo. 2024. EditScribe: Non-Visual Image Editing with Natural Language Verification Loops. ArXiv:2408.06632 [cs].

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *Preprint*, arXiv:2107.03374.

Paul Denny, Juho Leinonen, James Prather, Andrew Luxton-Reilly, Thezyrie Amarouche, Brett A. Becker, and Brent N. Reeves. 2023. Promptly: Using Prompt Problems to Teach Learners How to Effectively Utilize AI Code Generators. *arXiv preprint*. Issue: arXiv:2307.16364.

Jean-Baptiste Döderlein, Mathieu Acher, Djamel Eddine Khelladi, and Benoit Combemale. 2023. Piloting Copilot and Codex: Hot Temperature, Cold Prompts, or Black Magic? *arXiv preprint*. ArXiv:2210.14699 [cs].

Deborah Etsenake and Meiyappan Nagappan. 2024. Understanding the Human-LLM Dynamic: A Literature Survey of LLM Use in Programming Tasks. *arXiv preprint*. Version Number: 1.

Molly Q Feldman and Carolyn Jane Anderson. 2024. Non-Expert Programmers in the Generative AI Future. In *Proceedings of the 3rd Annual Meeting of the Symposium on Human-Computer Interaction for Work*, pages 1–19, Newcastle upon Tyne United Kingdom. ACM.

James Finnie-Ansley, Paul Denny, Brett A. Becker, Andrew Luxton-Reilly, and James Prather. 2022. The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming. In *Australasian Computing Education Conference*, ACE '22, pages 10–19, New York, NY, USA. Association for Computing Machinery. Event-place: Virtual Event, Australia.

Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J. Ericson, David Weintrop, and Tovi Grossman. 2023a. Studying the effect of AI Code Generators on Supporting Novice Learners in Introductory Programming. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–23, Hamburg Germany. ACM.

Majeed Kazemitabaar, Xinying Hou, Austin Henley, Barbara Jane Ericson, David Weintrop, and Tovi Grossman. 2023b. How Novices Use LLM-based Code Generators to Solve CS1 Coding Tasks in a Self-Paced Learning Environment. In *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research*, pages 1–12, Koli Finland. ACM.

Daniel Khashabi, Xinxi Lyu, Sewon Min, Lianhui Qin, Kyle Richardson, Sean Welleck, Hannaneh Hajishirzi, Tushar Khot, Ashish Sabharwal, Sameer Singh, and Yejin Choi. 2022. Prompt waywardness: The curious case of discretized interpretation

of continuous prompts. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 3631–3643, Seattle, United States. Association for Computational Linguistics.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Symposium on Operating Systems Principles (SOSP)*, pages 611–626, New York, NY, USA. Association for Computing Machinery.

Sam Lau and Philip Guo. 2023. From "Ban It Till We Understand It" to "Resistance is Futile": How University Programming Instructors Plan to Adapt as More Students Use AI Code Generation and Explanation Tools such as ChatGPT and GitHub Copilot. In *Proceedings of the 2023 ACM Conference on International Computing Education Research V.1*, pages 106–121, Chicago IL USA. ACM.

Michael Xieyang Liu, Advait Sarkar, Carina Negreanu, Benjamin Zorn, Jack Williams, Neil Toronto, and Andrew D. Gordon. 2023. "What It Wants Me To Say": Bridging the Abstraction Gap Between End-User Programmers and Code-Generating Large Language Models. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–31, Hamburg Germany. ACM.

Vivian Liu and Lydia B Chilton. 2022. Design Guidelines for Prompt Engineering Text-to-Image Generative Models. In *CHI Conference on Human Factors in Computing Systems*, pages 1–23, New Orleans LA USA. ACM.

AI @ Meta Llama Team. 2024. The Llama 3 Herd of Models. *arXiv preprint*.

Qianou Ma, Weirui Peng, Hua Shen, Kenneth Koedinger, and Tongshuang Wu. 2024. What You Say = What You Want? Teaching Humans to Articulate Requirements for LLMs. *arXiv preprint*. ArXiv:2409.08775 [cs].

Aman Madaan, Katherine Hermann, and Amir Yazdanbakhsh. 2023. What makes chain-of-thought prompting effective? a counterfactual study. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 1448–1535, Singapore. Association for Computational Linguistics.

Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. 2022. Rethinking the role of demonstrations: What makes in-context learning work? In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 11048–11064, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Asaf Achi Mordechai, Yoav Goldberg, and Reut Tsarfaty. 2024. NoviCode: Generating Programs from Natural Language Utterances by Novices. *arXiv preprint*. ArXiv:2407.10626 [cs].

Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2024. Reading Between the Lines: Modeling User Behavior and Costs in AI-Assisted Programming. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pages 1–16, Honolulu HI USA. ACM.

Syndey Nguyen, Hannah McLean Babe, Yangtian Zi, Arjun Guha, Carolyn Jane Anderson, and Molly Q Feldman. 2024. How Beginning Programmers and Code LLMs (Mis)read Each Other. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '24)*.

OpenAI. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

Jonas Oppenlaender. 2023. A taxonomy of prompt modifiers for text-to-image generation. *Behaviour & Information Technology*, pages 1–14.

James Prather, Brent N. Reeves, Paul Denny, Brett A. Becker, Juho Leinonen, Andrew Luxton-Reilly, Garrett Powell, James Finnie-Ansley, and Eddie Antonio Santos. 2023. "It's Weird That it Knows What I Want": Usability and Interactions with Copilot for Novice Programmers. *ACM Transactions on Computer-Human Interaction*, page 3617367.

James Prather, Brent N Reeves, Juho Leinonen, Stephen MacNeil, Arisoa S Randrianasolo, Brett A. Becker, Bailey Kimmel, Jared Wright, and Ben Briggs. 2024. The Widening Gap: The Benefits and Harms of Generative AI for Novice Programmers. In *Proceedings of the 2024 ACM Conference on International Computing Education Research - Volume 1*, pages 469–486, Melbourne VIC Australia. ACM.

Vipul Raheja, Dhruv Kumar, Ryan Koo, and Dongyeop Kang. 2023. CoEdIT: Text Editing by Task-Specific Instruction Tuning. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 5274–5291.

Hendrik Strobelt, Albert Webson, Victor Sanh, Benjamin Hoover, Johanna Beyer, Hanspeter Pfister, and Alexander M. Rush. 2022. Interactive and Visual Prompt Engineering for Ad-hoc Task Adaptation With Large Language Models. *IEEE Transactions on Visualization and Computer Graphics*, pages 1–11.

Tiffany Tseng, Ruijia Cheng, and Jeffrey Nichols. 2024. Keyframer: Empowering Animation Design using Large Language Models. *arXiv preprint*. Version Number: 1.

Annapurna Vadaparty, Daniel Zingaro, David H. Smith IV, Mounika Padala, Christine Alvarado, Jamie Gorson Benario, and Leo Porter. 2024. CS1-LLM: Integrating LLMs into CS1 instruction. In *Proceedings of the 2024 on Innovation and Technology in*

Computer Science Education V. 1, ITiCSE 2024, page 297–303, New York, NY, USA. Association for Computing Machinery.

Boshi Wang, Sewon Min, Xiang Deng, Jiaming Shen, You Wu, Luke Zettlemoyer, and Huan Sun. 2023. Towards understanding chain-of-thought prompting: An empirical study of what matters. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2717–2739, Toronto, Canada. Association for Computational Linguistics.

Albert Webson and Ellie Pavlick. 2022. Do prompt-based models really understand the meaning of their prompts? In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2300–2344, Seattle, United States. Association for Computational Linguistics.

Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. 2023. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. *arXiv preprint*. ArXiv:2302.11382 [cs].

Chunqiu Steven Xia, Yinlin Deng, and Lingming Zhang. 2024. Top leaderboard ranking = top coding proficiency, always? EvoEval: Evolving coding benchmarks via LLM. In *Conference on Language Modeling*.

Xi Ye and Greg Durrett. 2022. The unreliability of explanations in few-shot prompting for textual reasoning. In *Advances in Neural Information Processing Systems*.

Catherine Yeh, Gonzalo Ramos, Rachel Ng, Andy Huntington, and Richard Banks. 2024. GhostWriter: Augmenting Collaborative Human-AI Writing Experiences Through Personalization and Agency. *arXiv preprint*. ArXiv:2402.08855 [cs].

J.D. Zamfirescu-Pereira, Richmond Y. Wong, Bjoern Hartmann, and Qian Yang. 2023. Why Johnny Can't Prompt: How Non-AI Experts Try (and Fail) to Design LLM Prompts. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–21, Hamburg Germany. ACM.

Chen Zhu-Tian, Zeyu Xiong, Xiaoshuo Yao, and Elena Glassman. 2024. Sketch Then Generate: Providing Incremental User Feedback and Guiding LLM Code Generation through Language-Oriented Code Sketches. *arXiv preprint*. ArXiv:2405.03998 [cs].

A Dataset and Code Availability

The code and dataset for this submission is publicly available and licensed under the terms of the BSD 3 Clause license. The StudentEval dataset is licensed under under the terms of the OpenRAIL license.

The code and data for this paper are available at https://github.com/nuprl/substance-vs-style.

B Use of AI Assistants

Some authors used AI assistants while writing code for this paper.

C Computing Resources

The computational experiments for this paper were conducted with less than 1,000 hours of A100 GPU time. The models evaluated were Meta Llama 3.1 8B and 70B (Llama Team, 2024).

D Software Configuration

We use vLLM 0.6.2 for LLM inference (Kwon et al., 2023). We use spaCy 3.8.0 for lemmatization with the `en_core_web_trf` pipeline.

E Causal Analysis of Lexical Choices

This section describes the procedure we used to perform the causal analysis of lexical choices and presents detailed results.

E.1 Data Annotation Procedure

The overall approach to data annotation is described in §4.1. We provide some additional detail below.

The process for tagging concept references proceeded as follows. First, we developed an automated script to perform tagging automatically. This approximated the set of necessary tags, but a manual pass was necessary for numerous reasons. For instance, some student terms (e.g., convert) occurred in numerous problems, but were either function names or parameters in some. In other cases grammatical features, such as prepositions, led the automated approach to be insufficient (e.g., $takes:brings$ should be tagged as $takes:brings in$).

The two expert annotators, who are both CS1 instructors, then proceeded to perform a manual review. During this review, care was taken to tag idiosyncratic references; for instance, when a participant mistakenly referred to an input dictionary as a list, this was tagged under the dictionary category, so that we could explore substitutions of a more accurate term. The goal of this process was a consistent tag set, thus the annotators ultimately came to consensus on all tags for all prompts. Inter-annotator reliability was not calculated due to the emphasis on consensus and the number/precision of tags per prompt.

To gain insight into the range of terms used over problems, the annotators independently assessed two distinct prompts for each of the 48 problems, for a total of 96 problems. They then met to discuss their tagging edits. Out of this discussion, we made three main changes: (1) "given" was removed as a possible term, as it has too many possible use cases; (2) the Input concept was divided into the three lemmas of "parameters", "take", and "provide;" and (3) specific disambiguation for "concatenate" and "insert" was developed. The annotators then came to consensus on all tags for the 96 problems.

After this process, the above changes were made to the automatic tagging script and then the two annotators independently tagged the remainder of the problems in the dataset. They then met to discuss the tagging edits and determine the consensus decision. Most disagreements were easily resolved (e.g., missed tags, typos). The main substantive disagreement was regarding tags relevant to the String concept. Specifically, determining student meaning of character versus string was too challenging to tag consistently. Therefore, most mentions of character/s were removed from the String tag set. This was done retroactively to the original 96 problems as well.

E.2 Concepts, Expressions, and Interventions

Table 2 shows the lemmas for each concept category used in the lexical substitution experiments, along with the set of replacement terms and example expressions that students use to refer to them.

Concept	Lemma	Substitution Lemmas	Example Student Terms
String	string	word, phrase, string, character, set of characters	string, word, string of text, word or sentence, string of characters
List	list	brackets, set of brackets, set, list, array list, array	list, array, set, arrangement,series, collection, sequence
Dictionary	dictionary	map, dictionary	dictionary, dict, object, array
Integer	integer	integer, whole number, int	int, integer numbers, whole number
Key	key	key, item, entry, attribute, part, element, variable	key, key value, category element, variable, parameter
Input	parameter	parameter, argument, value provided, input	input, parameter, value, component, input value, value inputted
	take	take, bring in, accept, get, input	take, take in, take input of, get input
	provide	provide, enter, input	
Loop	loop through	go through, run through, iterate through, loop through, run a for loop through, look through, execute a for loop with	loop, loop through, go through, parse, iterate through, run through
Output	return	return, output, print, produce, display	return, output, print, provide, out put
Concatenate	concatenate	concatenate, combine, splice, add	concatenate, append, add, combine
Insert	insert	insert, add, append, attach	put, insert, input, add, give
Skip	skip	skip, avoid, neglect, ignore, remove	skip, ignore, avoid, neglect
Typecast	typecast	typecast, type cast, cast, convert, change	typecast, convert, turn, change

Table 2: Concepts, Lemmas, and Substitution Terms for Causal Analysis Experiments

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-4.4	0.69	-6.3	**<0.0001**
character	-1.1	0.32	-3.6	**0.0004**
phrase	-0.69	0.2	-3.4	**0.0006**
set of characters	-1.4	0.36	-4.1	**<0.0001**
string	0.048	0.054	0.87	0.38
word	-0.62	0.19	-3.3	**0.0009**

Table 3: Llama 8B mixed-effects model for String concept.

E.3 Experimental Method

For generations, we generated 200 completions for each model with temperature (0.2), top-p sampling (0.95), and a 512 token limit.

E.4 Statistical Analysis

Statistical significance results are from mixed-effects binary logistic regression models that include random effects for prompt ID and problem. The random effects structure for problem contains both random slopes and intercepts; due to issues with convergence, the random effects for prompt ID contain only random intercepts.

The outcome variable is the pass@1 rate calculated with 200 samples. All models were fit in R using the lme4 library (Bates et al., 2015) with sample weights of 200 (the number of observations from which the proportion was computed).

E.4.1 Type Concepts

Tables 3-16 provide the full mixed-effects results for datatype concepts.

E.4.2 Control Flow Concepts

Tables 18-31 provide the full mixed-effects results for control flow concepts.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-3.4	0.62	-5.5	**<0.0001**
character	-0.59	0.19	-3	**0.002**
phrase	-0.29	0.18	-1.6	0.1
set of characters	-1.1	0.22	-4.8	**<0.0001**
string	0.11	0.075	1.5	0.14
word	-0.098	0.12	-0.79	0.43

Table 4: Llama 70B mixed-effects model for String concept.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-0.14	1.4	-0.1	0.92
character	-1.7	0.54	-3.2	0.0013
phrase	-0.92	0.53	-1.7	0.083
set of characters	-4.3	0.79	-5.4	**<0.0001**
string	0.21	0.23	0.91	0.36
word	-0.18	0.35	-0.5	0.62

Table 5: Gpt-4o mini mixed-effects model for String concept.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-5.5	0.75	-7.4	**<0.0001**
array	0.1	0.087	1.2	0.24
array list	0.11	0.11	1	0.31
brackets	-1	0.23	-4.4	**<0.0001**
list	-0.0032	0.041	-0.078	0.94
set	-2.4	0.51	-4.7	**<0.0001**
set of brackets	-1.9	0.37	-5.2	**<0.0001**

Table 6: Llama 8B mixed-effects model for List concept.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-5.1	0.74	-6.9	**<0.0001**
array	-0.057	0.083	-0.69	0.49
array list	0.11	0.098	1.1	0.28
brackets	-0.55	0.16	-3.3	**0.0009**
list	-0.074	0.057	-1.3	0.19
set	-2.3	0.55	-4.1	**<0.0001**
set of brackets	-1.6	0.41	-3.9	**0.0001**

Table 7: Llama 70B mixed-effects model for List concept.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-4	1.4	-2.9	0.004
array	0.2	0.26	0.76	0.45
array list	0.25	0.23	1.1	0.29
brackets	-1.9	0.71	-2.7	0.006
list	0.24	0.082	2.9	0.0043
set	-4	0.89	-4.5	**<0.0001**
set of brackets	-2.9	1	-2.9	0.0036

Table 8: Gpt-4o mini mixed-effects model for List concept.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-5.9	0.9	-6.6	**<0.0001**
int	-0.1	0.091	-1.2	0.25
integer	0.057	0.038	1.5	0.13
whole number	-0.45	0.23	-1.9	0.052

Table 9: Llama 8B mixed-effects model for Integer concept.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-5.6	1.1	-5.3	**<0.0001**
int	-0.096	0.14	-0.71	0.48
integer	0.14	0.18	0.77	0.44
whole number	-0.1	0.21	-0.5	0.62

Table 10: Llama 70B mixed-effects model for Integer concept.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-3.8	4.1	-0.92	0.36
int	-0.18	0.42	-0.42	0.68
integer	-0.11	0.47	-0.22	0.82
whole number	-1.4	0.54	-2.7	0.0078

Table 11: Gpt-4o mini mixed-effects model for Integer concept.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-13	1.3	-9.9	**<0.0001**
dictionary	-0.099	0.056	-1.8	0.075
map	-0.066	0.41	-0.16	0.87

Table 12: Llama 8B mixed-effects model for Dictionary concept.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-12	1.4	-9	**<0.0001**
dictionary	-0.1	0.14	-0.73	0.46
map	-0.33	0.27	-1.2	0.23

Table 13: Llama 70B mixed-effects model for Dictionary concept.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-17	5.4	-3.2	0.0015
dictionary	1	0.54	1.9	0.063
map	1.1	0.9	1.3	0.21

Table 14: Gpt-4o mini mixed-effects model for Dictionary concept.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-9.8	1.3	-7.8	**<0.0001**
attribute	-0.7	0.32	-2.2	**0.028**
element	-0.25	0.15	-1.7	0.087
entry	-0.39	0.14	-2.8	**0.0048**
item	-0.28	0.14	-2	**0.047**
key	0.046	0.15	0.3	0.77
part	-0.59	0.21	-2.8	**0.005**
variable	-0.56	0.14	-3.9	**<0.0001**

Table 15: Llama 8B mixed-effects model for Key concept.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-6.5	1.4	-4.7	**<0.0001**
attribute	-0.87	0.43	-2	**0.04**
element	-0.29	0.16	-1.8	0.065
entry	-0.28	0.19	-1.5	0.14
item	-0.28	0.22	-1.3	0.21
key	-0.16	0.11	-1.5	0.15
part	-0.48	0.36	-1.3	0.18
variable	-0.85	0.5	-1.7	0.092

Table 16: Llama 70B mixed-effects model for Key concept.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-0.91	3.3	-0.27	0.78
attribute	-4.1	1.7	-2.4	0.019
element	-2.9	1.7	-1.7	0.087
entry	-2.2	1	-2.2	0.031
item	-2.8	1.6	-1.8	0.077
key	0.35	0.39	0.89	0.37
part	-2.8	1.7	-1.6	0.11
variable	-2.8	1.3	-2.1	0.039

Table 17: Gpt-4o mini mixed-effects model for Key concept.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-5	0.62	-8.1	**<0.0001**
display	-1.3	0.25	-5.1	**<0.0001**
output	-0.14	0.13	-1.1	0.27
print	-2.9	0.37	-7.8	**<0.0001**
produce	-0.16	0.12	-1.3	0.2
return	0.11	0.061	1.8	0.068

Table 18: Llama 8B mixed-effects model for Return concept.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-4.9	0.56	-8.7	**<0.0001**
display	-0.87	0.28	-3	**0.002**
output	-0.071	0.17	-0.42	0.67
print	-2.8	0.41	-6.8	**<0.0001**
produce	0.21	0.16	1.3	0.18
return	0.0061	0.096	0.063	0.95

Table 19: Llama 70B mixed-effects model for Return concept.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-2.6	1	-2.6	0.01
display	-6.1	1.1	-5.5	**<0.0001**
output	0.34	0.53	0.64	0.52
print	-19	2.1	-9.1	**<0.0001**
produce	0.78	0.34	2.3	0.023
return	0.68	0.28	2.4	0.014

Table 20: Gpt-4o mini mixed-effects model for Return concept.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-11	2.1	-5	**<0.0001**
execute a for loop with	-3.5	0.59	-6	**<0.0001**
go through	-0.56	0.15	-3.8	**0.0001**
iterate through	0.0045	0.14	0.032	0.97
look through	-0.41	0.27	-1.6	0.12
loop through	-0.38	0.27	-1.4	0.16
run a for loop through	-2.9	0.48	-6	**<0.0001**
run through	-0.37	0.2	-1.8	0.067

Table 21: Llama 8B mixed-effects model for Loop concept.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-10	1.8	-5.8	**<0.0001**
execute a for loop with	-4.7	1.3	-3.5	**0.0005**
go through	-0.92	0.56	-1.6	0.1
iterate through	-1.3	0.33	-4.1	**<0.0001**
look through	-0.62	0.45	-1.4	0.16
loop through	-1.4	0.34	-4.2	**<0.0001**
run a for loop through	-1.7	0.52	-3.3	**0.001**
run through	-0.34	0.48	-0.71	0.48

Table 22: Llama 70B mixed-effects model for Loop concept.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-3	4.7	-0.65	0.52
execute a for loop with	0.22	1.1	0.2	0.84
go through	-0.99	1.1	-0.94	0.35
iterate through	-0.36	0.74	-0.48	0.63
look through	0.15	0.74	0.2	0.84
loop through	0.33	1	0.32	0.75
run a for loop through	-0.19	0.96	-0.19	0.85
run through	-0.017	0.75	-0.022	0.98

Table 23: Gpt-4o mini mixed-effects model for Loop concept.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-15	4.1	-3.6	**0.0004**
enter	0.31	0.36	0.87	0.38
input	0.079	0.24	0.33	0.74
provide	1.8	1.2	1.5	0.13

Table 24: Llama 8B mixed-effects model for Input - Provide lemma.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-15	6.5	-2.2	**0.03**
enter	-0.27	0.43	-0.63	0.53
input	0.29	0.49	0.6	0.55
provide	-1.1	0.42	-2.6	0.008

Table 25: Llama 70B mixed-effects model for Input - Provide lemma.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-19	18	-1.1	0.28
enter	1	1.5	0.68	0.49
input	0.61	2.6	0.23	0.82
provide	-0.31	2.2	-0.14	0.89

Table 26: Gpt-4o mini mixed-effects model for Input - Provide lemma.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-5.6	0.83	-6.7	**<0.0001**
argument	-0.045	0.1	-0.43	0.67
input	0.088	0.06	1.5	0.14
parameter	-0.095	0.11	-0.86	0.39
value provided	-0.17	0.13	-1.3	0.19

Table 27: Llama 8B mixed-effects model for Input - Parameter lemma.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-5.4	0.98	-5.5	**<0.0001**
argument	0.28	0.13	2.2	**0.03**
input	-0.031	0.094	-0.33	0.74
parameter	0.23	0.14	1.6	0.1
value provided	0.27	0.16	1.7	0.086

Table 28: Llama 70B mixed-effects model for Input - Parameter lemma.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-4.2	3.4	-1.2	0.22
argument	-0.26	0.69	-0.37	0.71
input	0.34	0.35	0.96	0.34
parameter	-0.62	0.64	-0.97	0.33
value provided	-0.47	0.81	-0.58	0.56

Table 29: Gpt-4o mini mixed-effects model for Input - Parameter lemma.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-5.7	0.97	-5.9	**<0.0001**
accept	-0.061	0.078	-0.79	0.43
bring in	-0.051	0.14	-0.38	0.71
get	0.0086	0.11	0.081	0.94
input	-0.15	0.1	-1.4	0.15
take	0.029	0.056	0.51	0.61

Table 30: Llama 8B mixed-effects model for Input - Take lemma.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-5.2	1.1	-4.6	**<0.0001**
accept	-0.024	0.14	-0.17	0.87
bring in	-0.22	0.22	-1	0.31
get	0.14	0.14	0.96	0.34
input	0.054	0.14	0.39	0.7
take	-0.066	0.12	-0.53	0.6

Table 31: Llama 70B mixed-effects model for Input - Take lemma.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	0.89	4.9	0.18	0.85
accept	0.17	0.3	0.58	0.56
bring in	-1.6	0.94	-1.7	0.088
get	-0.79	0.6	-1.3	0.19
input	-0.34	0.66	-0.52	0.6
take	0.2	0.51	0.39	0.7

Table 32: Gpt-4o mini mixed-effects model for Input - Take lemma.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-4.4	1.5	-2.8	**0.005**
add	0.11	0.27	0.39	0.7
combine	-0.14	0.13	-1.1	0.29
concatenate	0.24	0.14	1.7	0.081
splice	-0.56	0.19	-3	**0.003**

Table 33: Llama 8B mixed-effects model for Concatenate concept.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-1.5	1.5	-0.97	0.33
add	-0.024	0.21	-0.12	0.91
combine	0.37	0.43	0.87	0.38
concatenate	0.28	0.35	0.82	0.41
splice	-0.31	0.51	-0.61	0.55

Table 34: Llama 70B mixed-effects model for Concatenate concept.

E.4.3 Operation Concepts

Tables 33-43 provide the full mixed-effects results for control flow concepts.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	17	5.3	3.3	0.0011
add	9.8	5.8	1.7	0.089
combine	-4.2	1.9	-2.2	0.025
concatenate	-2.2	1	-2.2	0.03
splice	-2.6	1.2	-2.2	0.031

Table 35: Gpt-4o mini mixed-effects model for Concatenate concept.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-5.5	1.8	-3	**0.003**
add	-0.091	0.12	-0.73	0.47
append	-0.38	0.46	-0.82	0.41
attach	-0.53	0.45	-1.2	0.24
insert	-1.4	1.1	-1.3	0.18

Table 36: Llama 8B mixed-effects model for Append concept.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-9.9	1.9	-5.3	**<0.0001**
add	-0.4	0.22	-1.8	0.067
append	-0.51	0.33	-1.5	0.12
attach	-0.11	0.34	-0.32	0.75
insert	-0.76	0.45	-1.7	0.089

Table 37: Llama 70B mixed-effects model for Append concept.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-3.6	4.3	-0.85	0.4
add	0.62	1	0.6	0.55
append	-0.88	0.84	-1	0.3
attach	0.071	0.8	0.088	0.93
insert	-0.063	1.3	-0.048	0.96

Table 38: Gpt-4o mini mixed-effects model for Append concept.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-13	3.8	-3.3	**0.001**
avoid	-0.76	0.28	-2.7	**0.006**
ignore	0.014	0.17	0.083	0.93
neglect	-0.18	0.28	-0.62	0.54
remove	-4.2	2.1	-2	**0.046**
skip	-0.21	0.46	-0.46	0.65

Table 39: Llama 8B mixed-effects model for Skip concept.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-14	6.4	-2.2	**0.03**
avoid	0.041	0.81	0.051	0.96
ignore	-0.98	0.22	-4.4	**<0.0001**
neglect	-1.2	0.43	-2.7	**0.007**
remove	-6.6	3.6	-1.8	0.068
skip	-0.66	0.47	-1.4	0.16

Table 40: Llama 70B mixed-effects model for Skip concept.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-17	12	-1.4	0.15
avoid	0.42	0.68	0.62	0.54
ignore	-3.3	1.6	-2	0.044
neglect	-3.5	1.8	-1.9	0.052
remove	1	2.8	0.36	0.72
skip	-7.1	3.7	-1.9	0.053

Table 41: Gpt-4o mini mixed-effects model for Skip concept.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-5	1.5	-3.3	**0.001**
cast	-0.86	0.39	-2.2	**0.028**
change	-1.5	0.85	-1.7	0.087
convert	-0.048	0.27	-0.18	0.86
type cast	-0.73	0.54	-1.4	0.17
typecast	-1.4	0.86	-1.6	0.1

Table 42: Llama 8B mixed-effects model for Typecast concept.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	-7.6	4.3	-1.8	0.074
cast	-0.89	0.74	-1.2	0.23
change	0.27	0.13	2	**0.045**
convert	0.31	0.23	1.3	0.19
type cast	-0.61	0.72	-0.85	0.4
typecast	-1	0.97	-1.1	0.28

Table 43: Llama 70B mixed-effects model for Typecast concept.

Fixed effects	$\widehat{\beta}$	SE	z	p
(Intercept)	0.59	11	0.055	0.96
cast	3.3	1.1	3.1	0.0022
change	-1.6	1.4	-1.1	0.26
convert	1.9	0.72	2.6	0.0082
type cast	4.9	2	2.4	0.017
typecast	5.4	1.5	3.6	0.00028

Table 44: Gpt-4o mini mixed-effects model for Typecast concept.

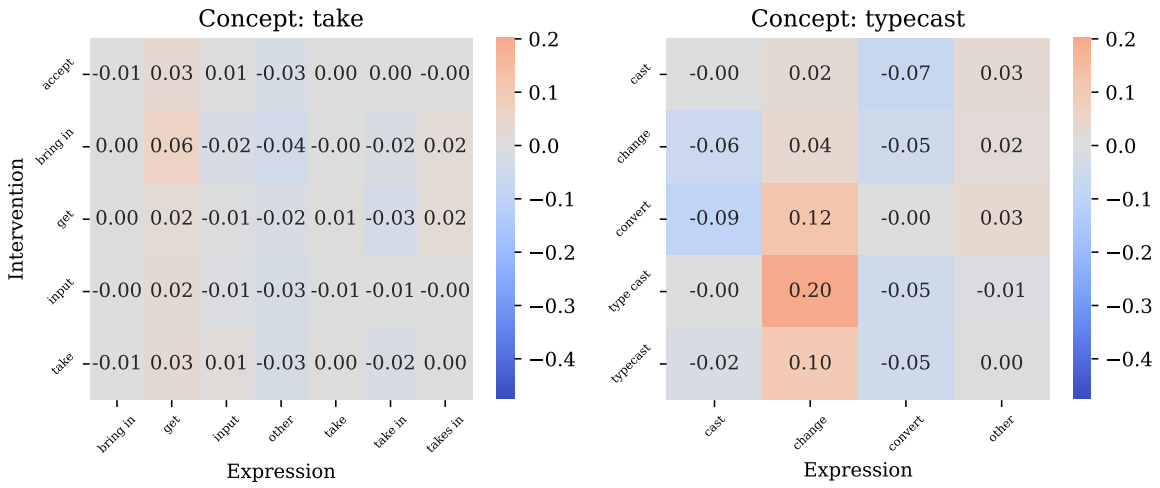

Figure 6: This heatmap shows the difference in pass rates (pass@1) using Meta Llama 3.1 8B after replacing the original expression of a concept in a prompt (x-axis) with a the expression chosen for the intervention (y-axis). We present one heatmap per concept. We report differences on the subset of prompts that have the original expression. We group rare expressions into a single *Other* class for each concept. See figures 9 and 10 for more categories.

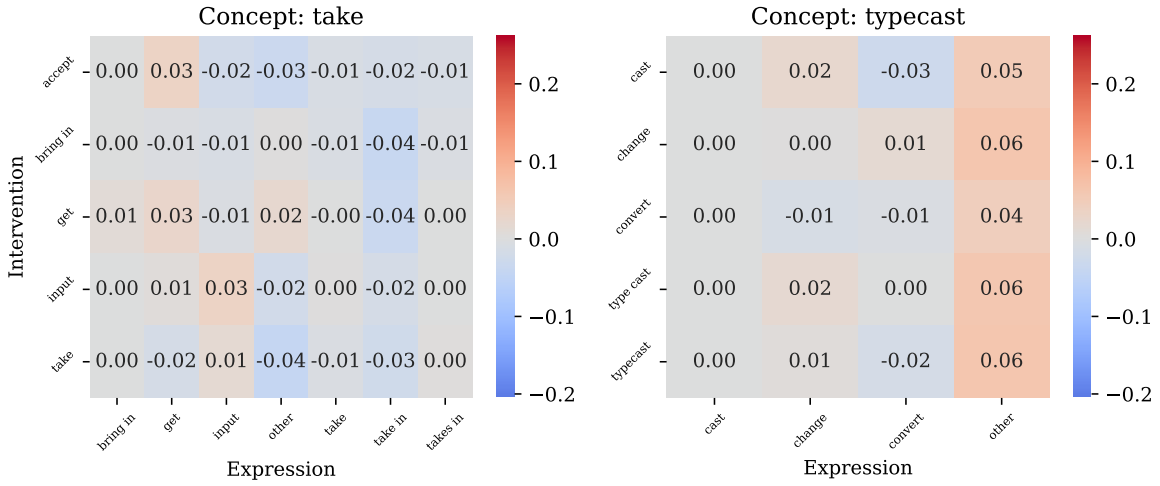

Figure 7: For Meta Llama 3.1 70B. See the caption for Figure 6 for more information.

E.5 Substitution Visualizations

Figures 6, 9 and 10 presents the results of causal interventions using Meta Llama 3.1 8B (Llama Team, 2024). Figures 7, 11 and 12 presents the results of causal interventions using Meta Llama 3.1 70B. Figures 8, 13 and 14 presents the results of causal interventions using OpenAI Gpt-4o mini.

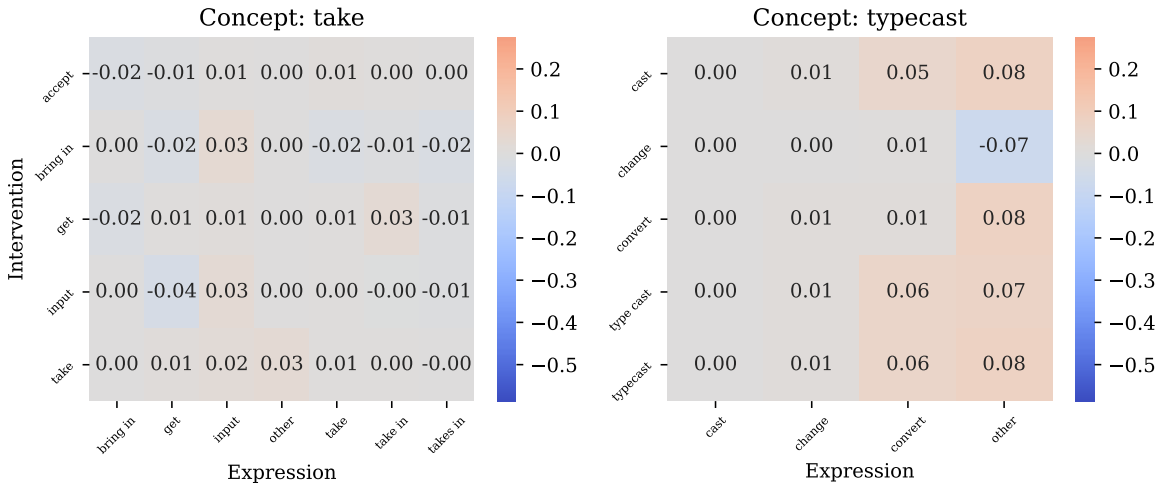

Figure 8: For Gpt-4o mini. See the caption for Figure 6 for more information.

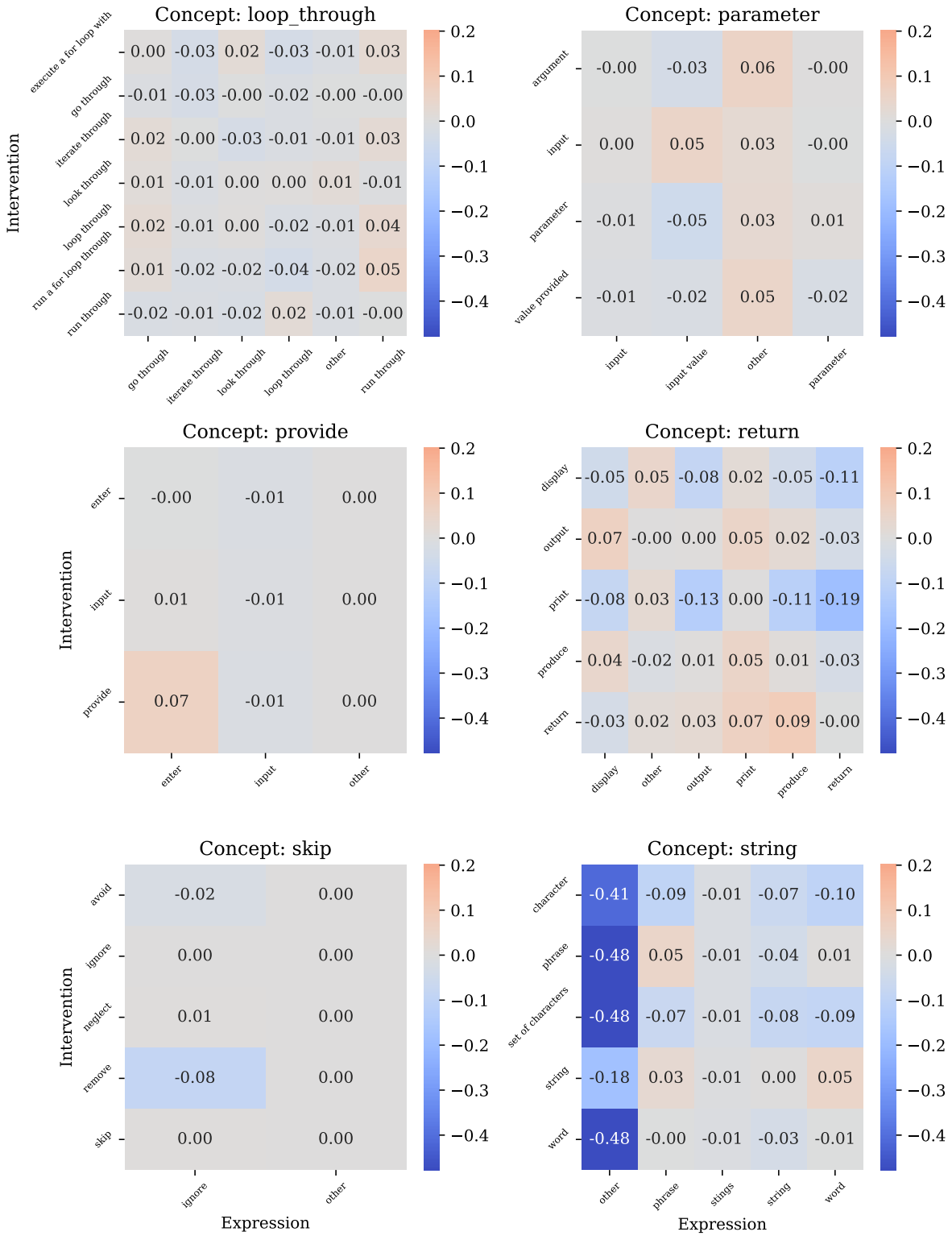

Figure 9: Continuation of Figure 6. See the caption of that figure for more information.

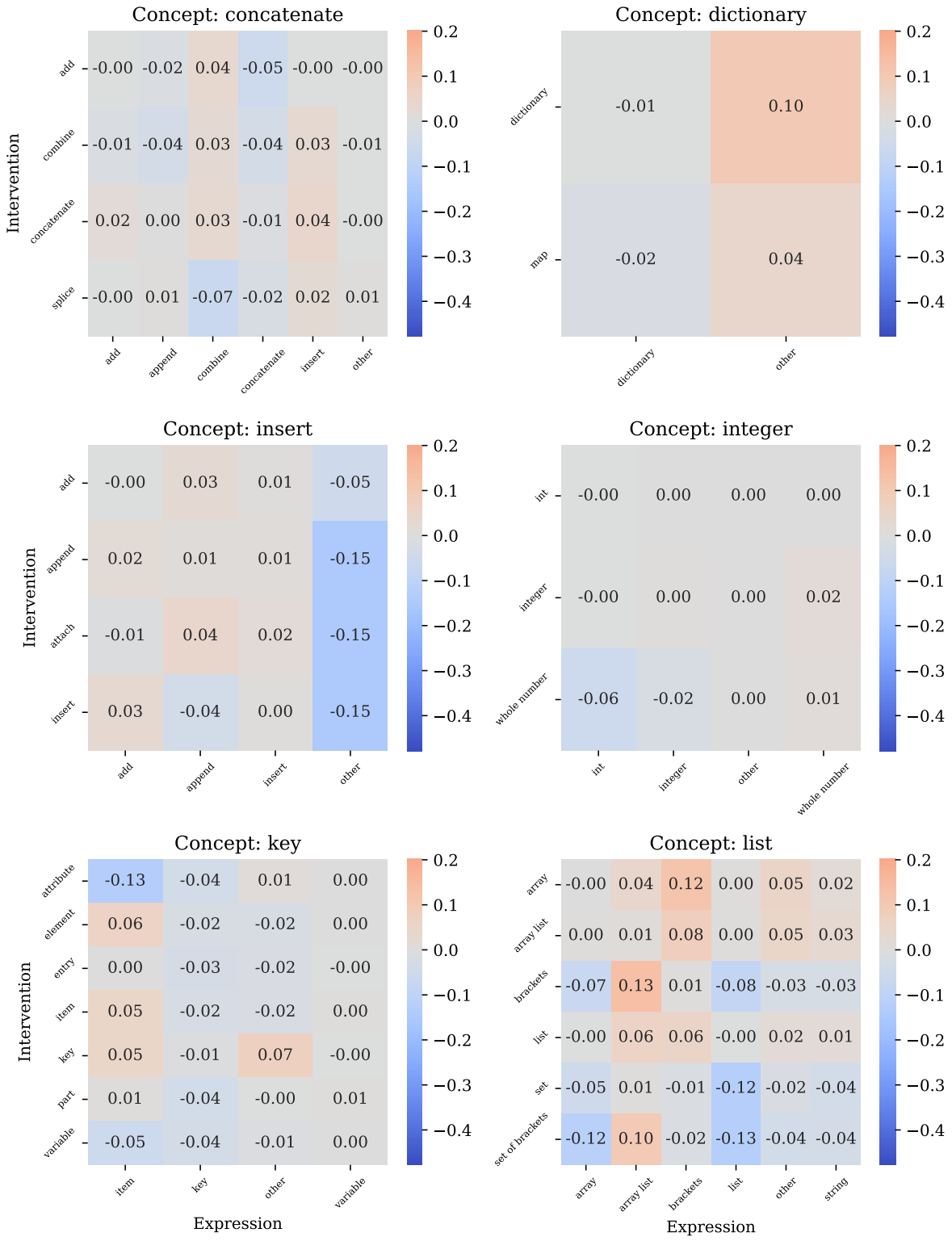

Figure 10: Continuation of Figure 6. See the caption of that figure for more information.

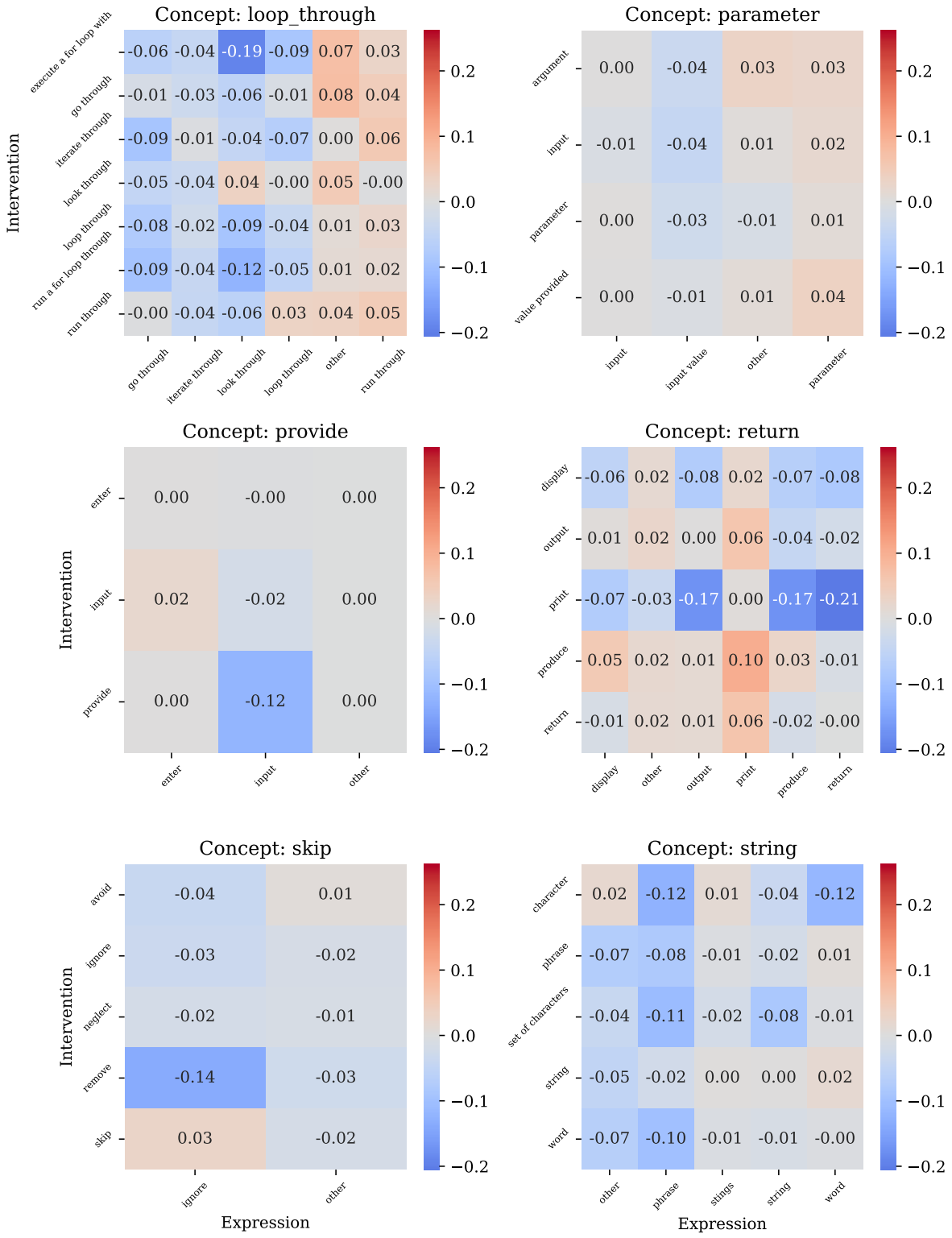

Figure 11: Continuation of Figure 7. See the caption of that figure for more information.

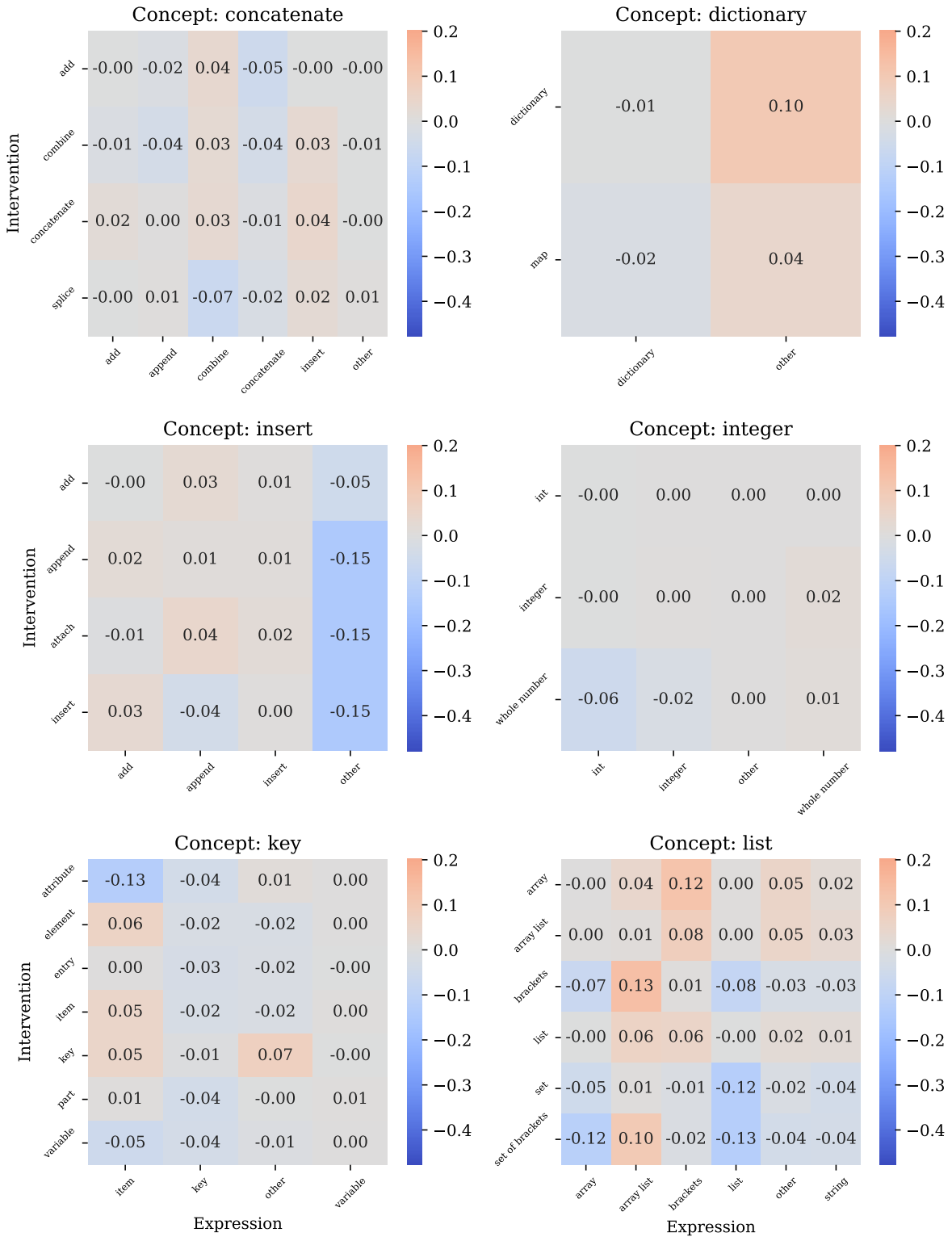

Figure 12: Continuation of Figure 7. See the caption of that figure for more information.

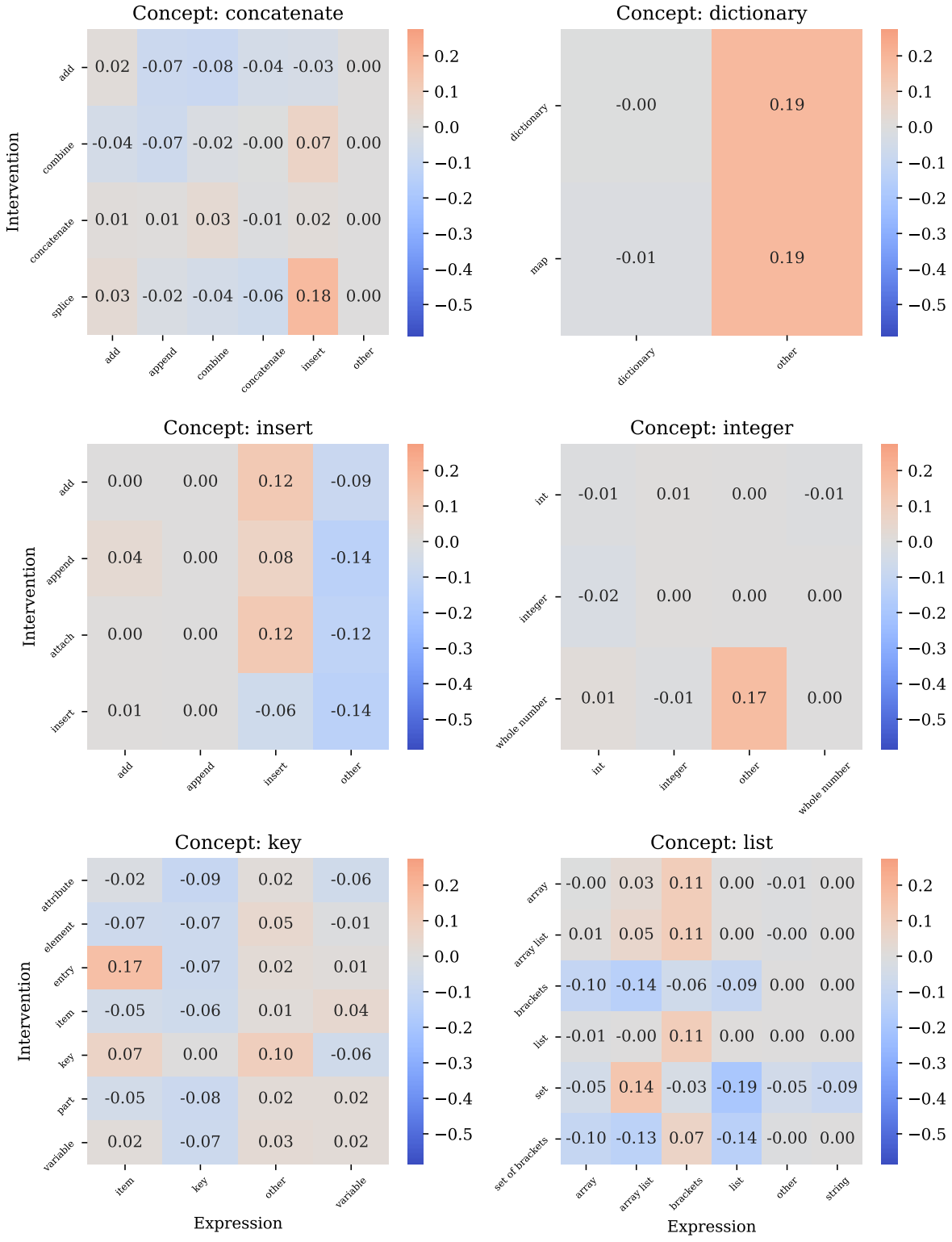

Figure 13: Continuation of Figure 8. See the caption of that figure for more information.

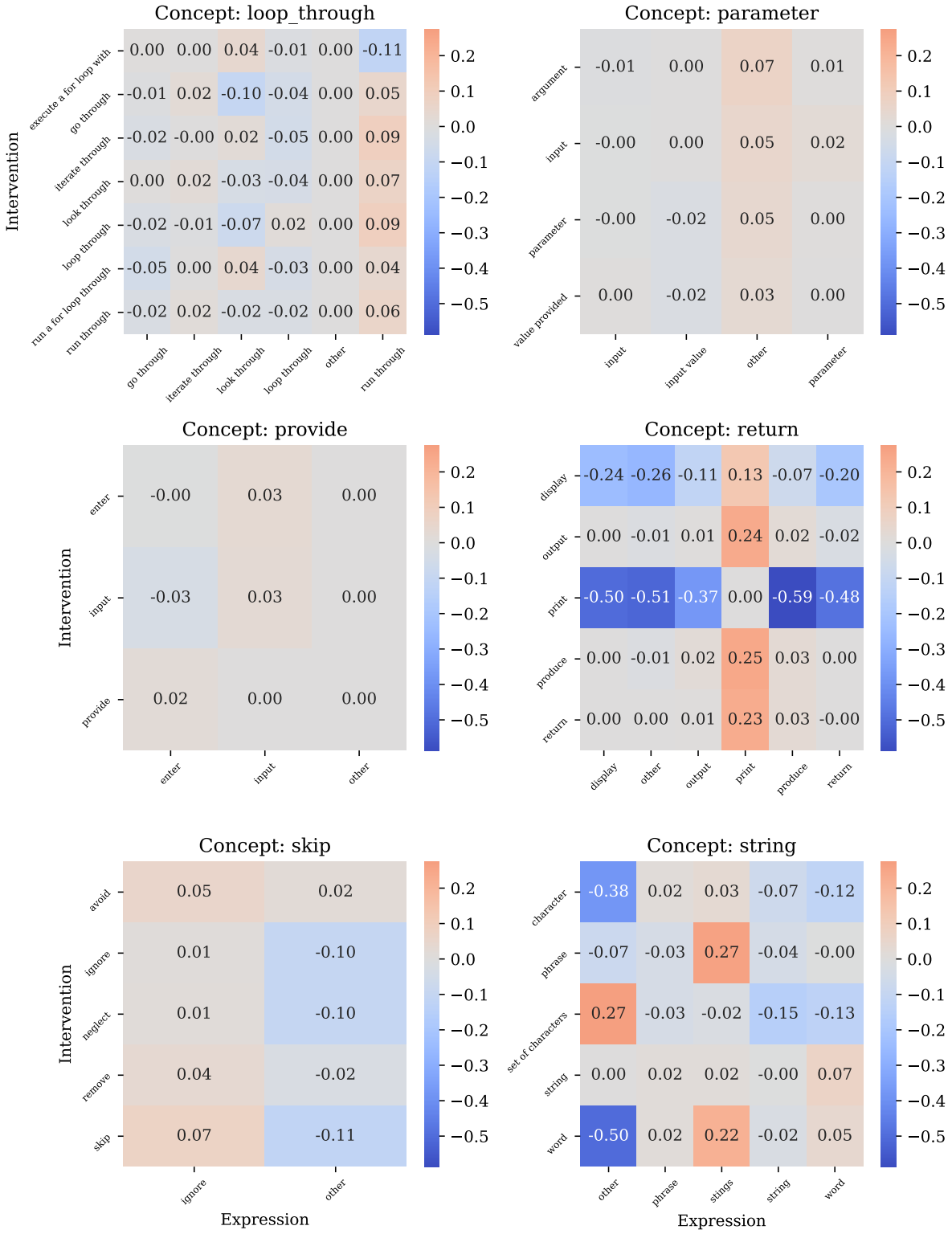

Figure 14: Continuation of Figure 8. See the caption of that figure for more information.

```
def altText(s):
 if len(s) == 1:
     return s.upper()
 else:
     return s[::2].upper() + s[1::2]
```

Figure 15: Reordering letters unexpectedly.

F Analyzing Prompt Trajectories

F.1 Tagging Prompt Clues

Four expert annotators tagged the information content of the 290 prompt trajectories. All annotators have experience teaching courses in Python programming and are therefore familiar with how students talk about programming concepts.

Annotators developed the sets of clues associated with problems by reading the successful prompts, the expert-written prompts from the original StudentEval dataset, and reflecting on the common information. Although information can be annotated at different levels of granularity, we strove for 3-8 clues per problem.

After annotation, a consistency check was performed on each prompt. Any inconsistencies in tagging (e.g., tagging an "add" operation for an existing clue; tagging a "modify" operation for a clue that had not previously been tagged) were corrected at this stage.

F.2 Additional Style Matters Examples

We examine additional examples where students include all necessary clues in their prompt, but the model's generated function still fails tests.

A common model error observed across two problems (`topScores` in Figure 16 and `sort_physicists` in Figure 17 and Figure 18) consists of a sorting error. Both problems receive as input a nested list, with the inner lists containing fixed elements: $[[x_0, ..., x_n], ..., [x_0, ..., x_n]]$. The problems stipulate that the generated function must return one of the elements x_i, sorted by another elements x_k, where $k \neq x$. The error the model consistently makes is filtering out the key required for sorting, then subsequently attempting to sort. This however cannot be done without the sorting key. Thus, the model often simply calls `sort`, eluding the key. One plausible explanation to why this happens is that human programmers are unlikely to delete the sorting key first, then try to sort. For this reason, training data may not include many examples of how to sort in this way. Note that in all students' subsequent successful attempts, the model deletes the sorting key *after* sorting.

In a prompt from student46 for the `planets_mass` problem (Figure 19), the model conflates an extra piece of information ("first letter capitalized") with the definition of a planet. Removing this single line leads the student to success. These examples serve to illustrate the kind of ambiguity in the wording of a prompt which can make the difference between success and fail.

F.3 Clue Sets

Here we provide the clue sets for all 33 problems.

Problem: `add_int`
Signature: `def add_int(lst, num):`
Clues:

1. edge case of list in list
2. concatenate num to strings
3. add num to integers
4. return list

Problem: `add_up`
Signature: `def add_up(arr):`
Clues:

1. 2D array

2. sum integer
3. sum float
4. return the sum of all elements
5. mention 0 base case
6. misdirection - add number within string

Problem: `altText`
Signature: `def altText(s):`
Clues:
1. input string
2. alternating uppercase
3. return all letters, including spaces
4. first letter upper

Problem: `assessVowels`
Signature: `def assessVowels(s):`
Clues:
1. argument s is a string
2. result is a list of strings
3. result is the vowels present in the argument
4. result has both upper and lower case vowels

Problem: `changeSection`
Signature: `def changeSection(s,i):`
Clues:
1. result is a string
2. result reverses a part of the argument 's'
3. the result reverses the first 'i' characters of the argument
4. the result also includes the remaining characters of 's', but not reversed

Problem: `check_prime`
Signature: `def check_prime(num):`
Clues:
1. convert input string to int
2. output bool
3. check prime
4. correct description of a procedure to check prime number

Problem: `combine`
Signature: `def combine(l1,l2):`
Clues:
1. input 2 lists
2. row correspondence
3. output 1 2d array

Problem: `convert`
Signature: `def convert(lst):`
Clues:
1. takes a list of numbers
2. maps numbers to letters
3. joins letters
4. -1 means split
5. return list of strings

Problem: `create_list`
Signature: `def create_list(dt, lst):`
Clues:
1. takes a dict and a list
2. looks up list items in dict

3. construct list with matching values
4. use None for items that aren't in dict
5. return list

Problem: `fib`
Signature: `def fib(n):`
Clues:

1. check if a Fib number
2. returns a Boolean
3. explanation of Fib
4. construct set of Fib numbers
5. hardcodes numbers
6. bound set

Problem: `findHorizontals`
Signature: `def findHorizontals(puzzle,wordList):`
Clues:

1. input is two lists
2. find words in second list within strings in first list
3. return dictionary
4. keys are words
5. values are indices of strings where words are found
6. words can be backwards or forwards

Problem: `find_multiples`
Signature: `def find_multiples(start,stop,factor):`
Clues:

1. return multiples
2. inclusive start and stop

Problem: `generateCardDeck`
Signature: `def generateCardDeck(suits, vals):`
Clues:

1. takes two lists
2. creates all pairs from the lists
3. sort alphabetically
4. first list item comes before second list item in pairs
5. return list

Problem: `getSeason`
Signature: `def getSeason(month):`
Clues:

1. input is string
2. month to season
3. return lowercase
4. explain which are which

Problem: `increaseScore`
Signature: `def increaseScore(score):`
Clues:

1. input integer
2. if less than 10, make 10
3. if 10 or more, add 1
4. if negative, turn positive
5. if single digit, add 0
6. return

Problem: `laugh`
Signature: `def laugh(size):`

Clues:

1. prefix h
2. reverse order
3. number of a's is based on size
4. space separation
5. down to 1
6. repetition
7. misdirection-print instead of return

Problem: `pattern`

Signature: `def pattern(value):`

Clues:

1. takes an int
2. produces a nested list
3. there are value n of inner lists
4. each inner list is from 1 to value
5. returns

Problem: `percentWin`

Signature: `def percentWin(guess,answers):`

Clues:

1. takes two lists
2. compares items from both lists and counts matches
3. computes percent match
4. rounds to whole percent
5. convert to string and add "
6. returns

Problem: `planets_mass`

Signature: `def planets_mass(planets):`

Clues:

1. takes a dictionary
2. skip Pluto
3. skip Sun
4. look up in dictionary
5. sum masses
6. return

Problem: `print_time`

Signature: `def print_time(day,hour):`

Clues:

1. input is a string and an int
2. how to distinguish sleeping
3. how to distinguish weekday versus weekend
4. short form of day
5. return not print

Problem: `readingIceCream`

Signature: `def readingIceCream(lines):`

Clues:

1. input is a list of strings
2. go through all strings
3. split on tab
4. extract last item from each string
5. convert to float
6. sum numbers
7. return total

Problem: `remove_odd`
Signature: `def remove_odd(lst):`
Clues:
1. takes a (potentially mixed) list of numbers
2. removes only odd numbers
3. removes only integers
4. returns list

Problem: `reverseWords`
Signature: `def reverseWords(words):`
Clues:
1. takes a list of strings
2. reverses each word in list
3. sorts list
4. reverse before sort
5. returns list

Problem: `set_chars`
Signature: `def set_chars(s,c,l):`
Clues:
1. input is described correctly
2. second argument is used to replace certain characters
3. third argument contains list of indices to replace
4. return string
5. handle indices outside string length

Problem: `sortBySuccessRate`
Signature: `def sortBySuccessRate(nominations):`
Clues:
1. input is list of dictionaries
2. add a key success
3. success is wins/noms
4. round success
5. sort by success
6. return

Problem: `sort_physicists`
Signature: `def sort_physicists(scientists):`
Clues:
1. Input is a list of lists
2. specify inner list structure
3. filter list with the right key
4. sort list with the right key
5. specify return
6. sort

Problem: `sortedBooks`
Signature: `def sortedBooks(books, writer):`
Clues:
1. takes a list of dictionaries
2. takes an author
3. removes books not by that author
4. sorts list
5. sorts list by year
6. returns list

Problem: `student_grades`
Signature: `def student_grades(students, grades):`

Clues:

1. input is two dictionaries
2. match keys to values between dictionaries
3. create a new dictionary with lists of grades
4. return

Problem: `subtract_add`

Signature: `def subtract_add(lst):`

Clues:

1. input is list
2. convert items to integers
3. alternate add and subtract
4. add is even indices and subtract is odd indices
5. return result

Problem: `times_with`

Signature: `def times_with(meetings):`

Clues:

1. input is nested list
2. describe nested list structure
3. make dictionary where keys are people and values are numbers
4. sum times
5. return dictionary

Problem: `topScores`

Signature: `def topScores(lst):`

Clues:

1. nested list with items in format [grade, name]
2. return only names with grade >80
3. return list of names
4. sort by grades descending
5. only top 5 scoring names
6. equal 80 edge case

Problem: `total_bill`

Signature: `def total_bill(grocery_list, sales_tax):`

Clues:

1. input is a list
2. list structure explained
3. input is sales tax
4. multiply item by its quantity
5. sum items
6. apply sales tax
7. round
8. return total

Problem: `translate`

Signature: `def translate(lexicon,sentence):`

Clues:

1. input 1 is a dictionary
2. input 2 is a string
3. extract words from string
4. look up words in dictionary
5. use UNK for words that are not in dictionary
6. function returns
7. join string

```
def topScores(lst):
    """
    taken lst, loop each list and if the zeroth index is more than or equal to 80,
    add
    the 1st index into a list and return this new list in the order of the highest
    zeroth index. If the list is larger than 4 items, then return only 4 items with
    the highest zeroth index.
    """

    new_lst = []
    for i in lst:
        if i[0] >= 80:
            new_lst.append(i[1])
    new_lst.sort(reverse=True)
    if len(new_lst) > 4:
        return new_lst[:4]
    else:
        return new_lst

def topScores(lst):
    """
    if 0th index of the nested list of lst is greater than or equal to 80, return a
    list of the 1st index in the order from the largest 0th index integer. Return up
    to a list of 5 items.
    """

    return [x[1] for x in sorted(lst, key=lambda x: x[0], reverse=True) if x[0] >=
    80][:5]
```

Figure 16: Student53 example prompt for topScores. The second prompt is the student's final successful attempt.

```
def sort_physicists(scientists):
    """
    The internal lists contain name, year graduated and subject. If the subject is
    "Physics" return just the names in order of what year they graduated from lowest
    to highest
    """

    physicists = []
    for scientist in scientists:
        if scientist[2] == "Physics":
            physicists.append(scientist[0])
    physicists.sort()
    return physicists

def sort_physicists(scientists):
    """
    The internal lists contain name, year graduated and subject. If the subject is
    "Physics" return just the names, in order of what year they graduated from
    lowest
    to highest
    """

    return [name for name, year, subject in sorted(scientists, key=lambda x: x[1])
    if subject == "Physics"]
```

Figure 17: Attempt by student36 at sort_physicists. The second prompt is the student's final successful attempt.

```
def sort_physicists(scientists):
    """
    Given a set of lists with a scientist, number, and a subject, respectively, only
    return the scientist if the subject is "Physics" in a list. Then sort the list
    into ascending order based on the number, but don't return the number.
    """

    physicists = []
    for scientist in scientists:
        if scientist[2] == "Physics":
            physicists.append(scientist[0])
    physicists.sort()
    return physicists

def sort_physicists(scientists):
    """
    Given a set of lists with a scientist, number, and a subject, respectively, only
    return the scientist if the subject is "Physics" in a list. Then sort the
    scientists based on the numbers given in the set of lists from smallest to
    largest. For example, if the scientist has the smallest number attached to them
    in
    the set of lists, they should be placed first in the output list.
    """

    physicists = []
    for scientist in scientists:
        if scientist[2] == "Physics":
            physicists.append(scientist[0])
    physicists.sort(key=lambda x: x[1])
    return physicists
```

Figure 18: Student77's attempt at `sort_physicists`. The second prompt is the student's final successful attempt.

```
def planets_mass(planets):
 """
 The function will take in a dictionary
 where the keys are strings, with the first letter capitalized, and integers
 are stored inside them. It will remove
 any strings that are not associated
 with planets, then it will take all
 the integers stored inside and add
 them together and output that.
 """
Completion 1
 for key in planets:
     if key[0].isupper() == False:
         del planets[key]
 mass = 0
 for key in planets:
     mass += planets[key]
 return mass

 Completion 2
 total = 0
 for planet in planets:
     if planet == "Mercury" or planet == \
     "Venus" or planet == "Earth" or \
     planet == "Mars" or planet == \
     "Jupiter" or planet == "Saturn" or \
     planet == "Uranus" or \
     planet == "Neptune":
         total += planets[planet]
 return total
```

Figure 19: two attempts by student46 at `planets_mass`. Deleting the terms in blue results in the second completion

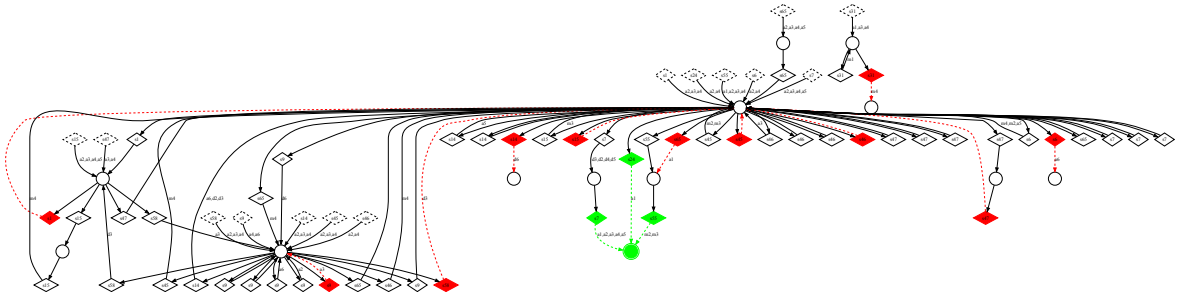

Figure 20: Prompt trajectories for the “add up” problem.

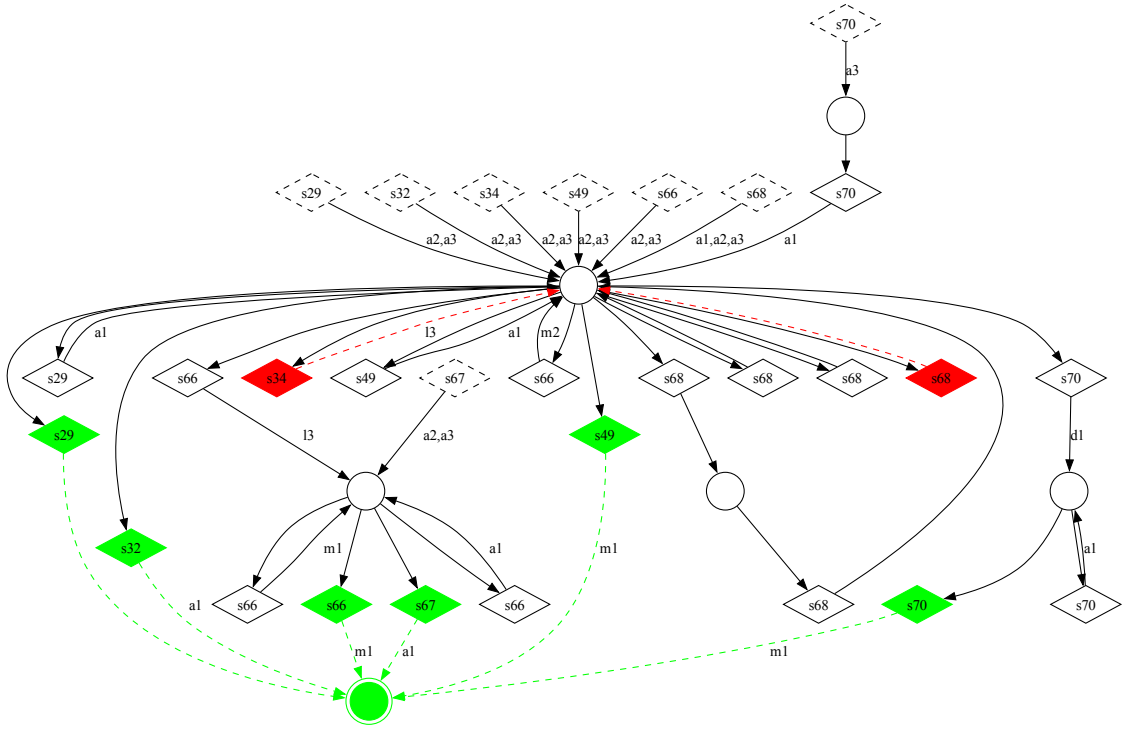

Figure 21: Prompt trajectories for the “check prime” problem.

F.4 All Prompt Trajectory Graphs

The prompt trajectories for all remaining problems are in Figure 20—Figure 51.

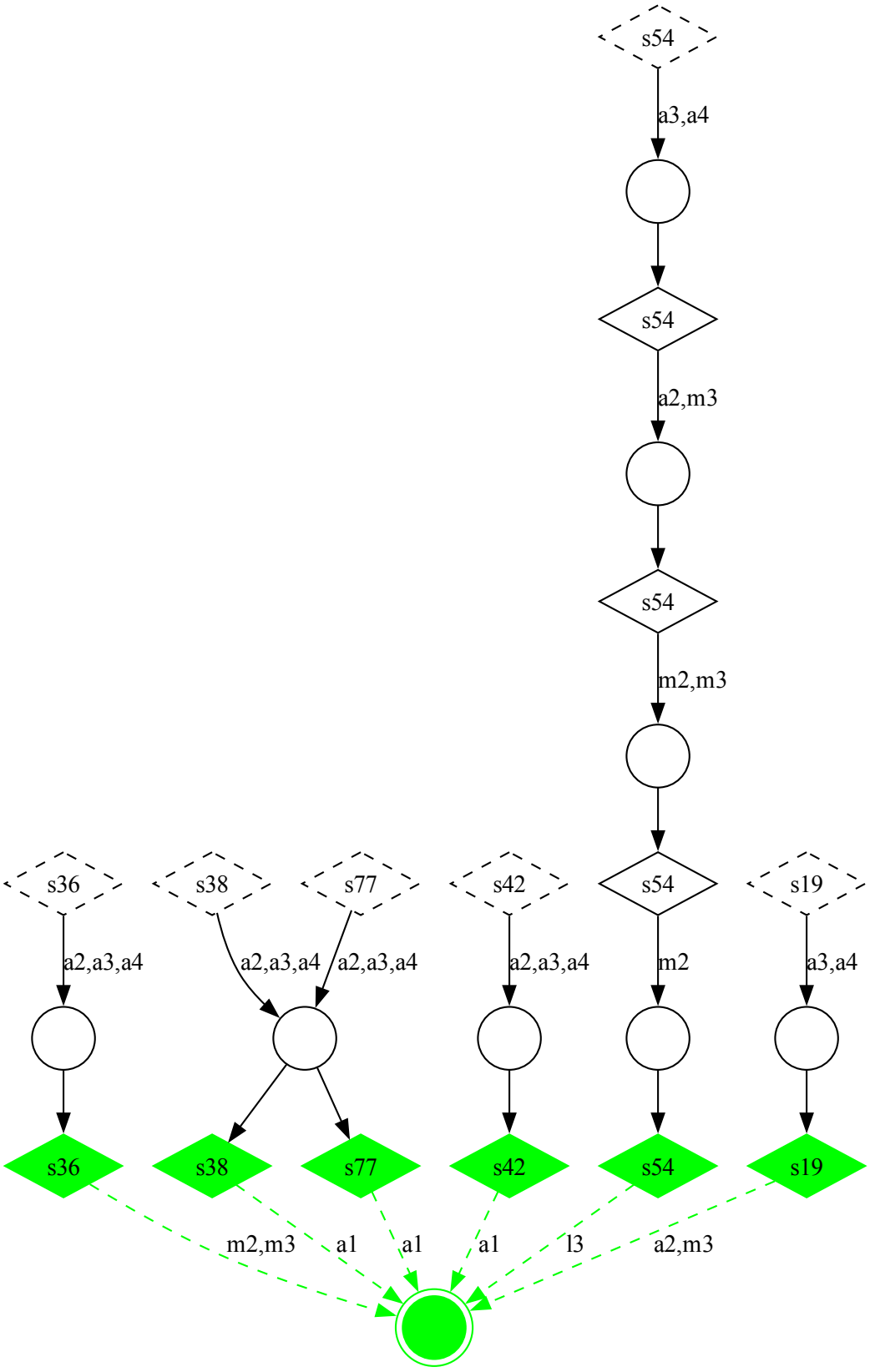

Figure 22: Prompt trajectories for the “add int” problem.

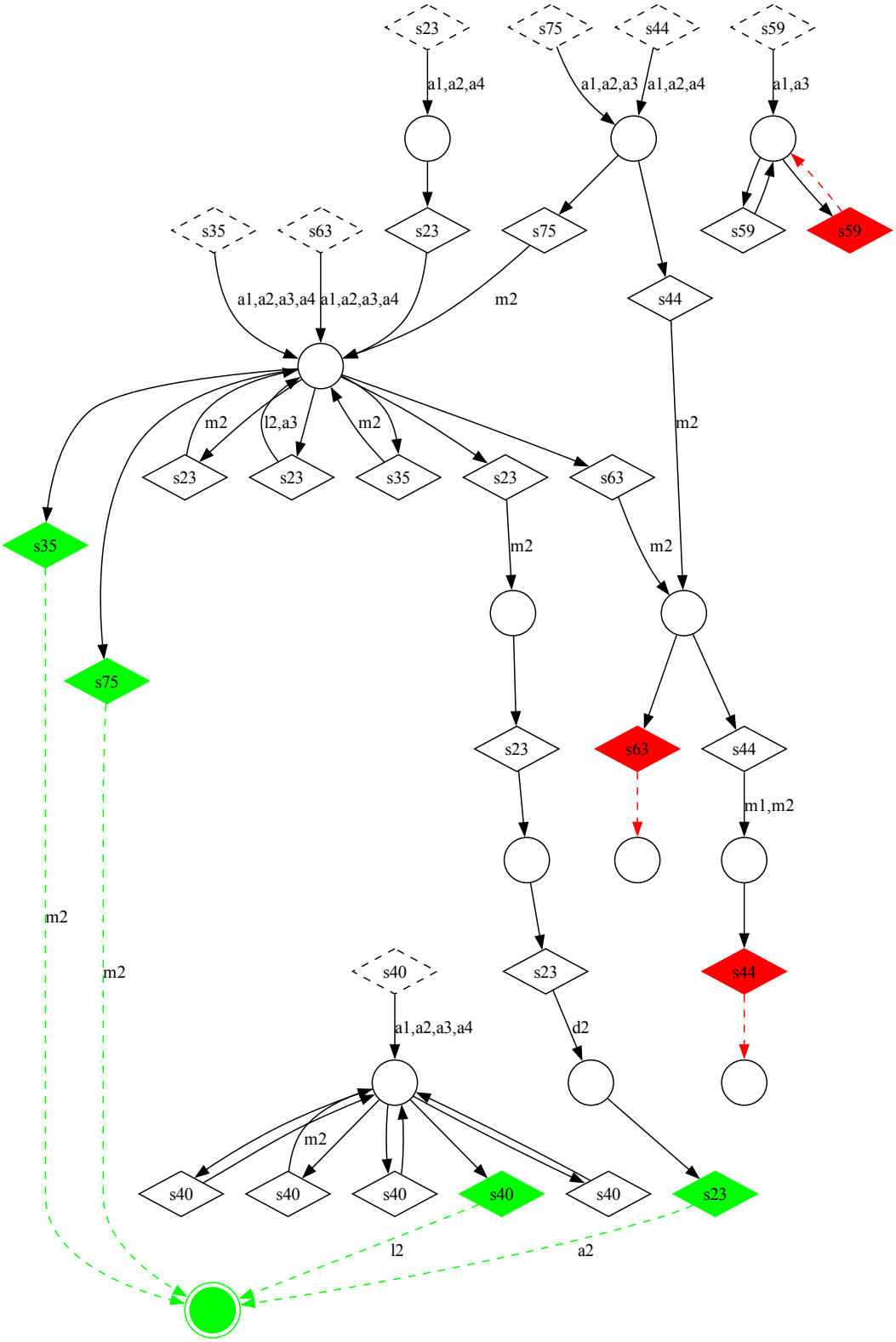

Figure 23: Prompt trajectories for the “altText” problem.

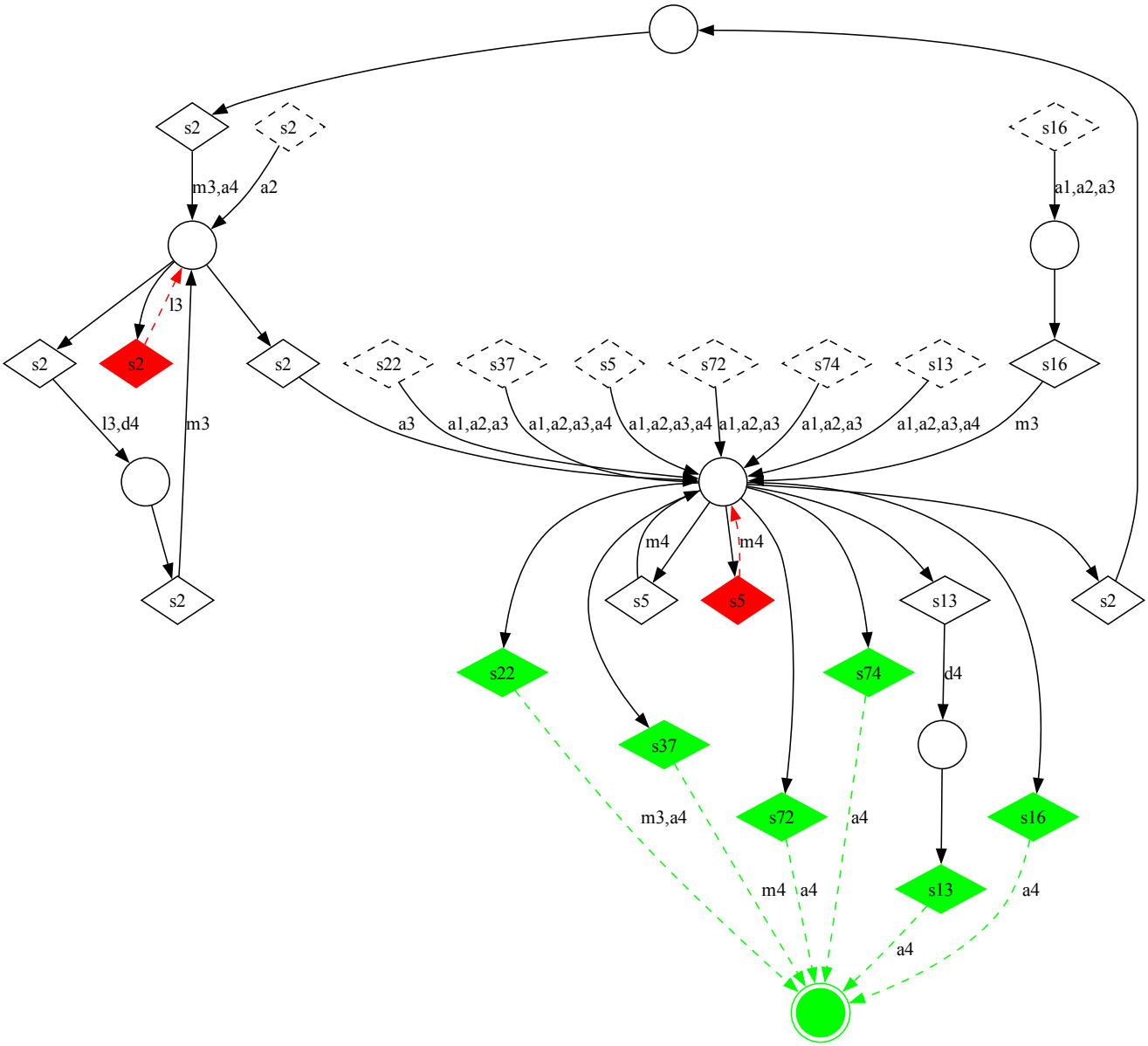

Figure 24: Prompt trajectories for the “assessVowels” problem.

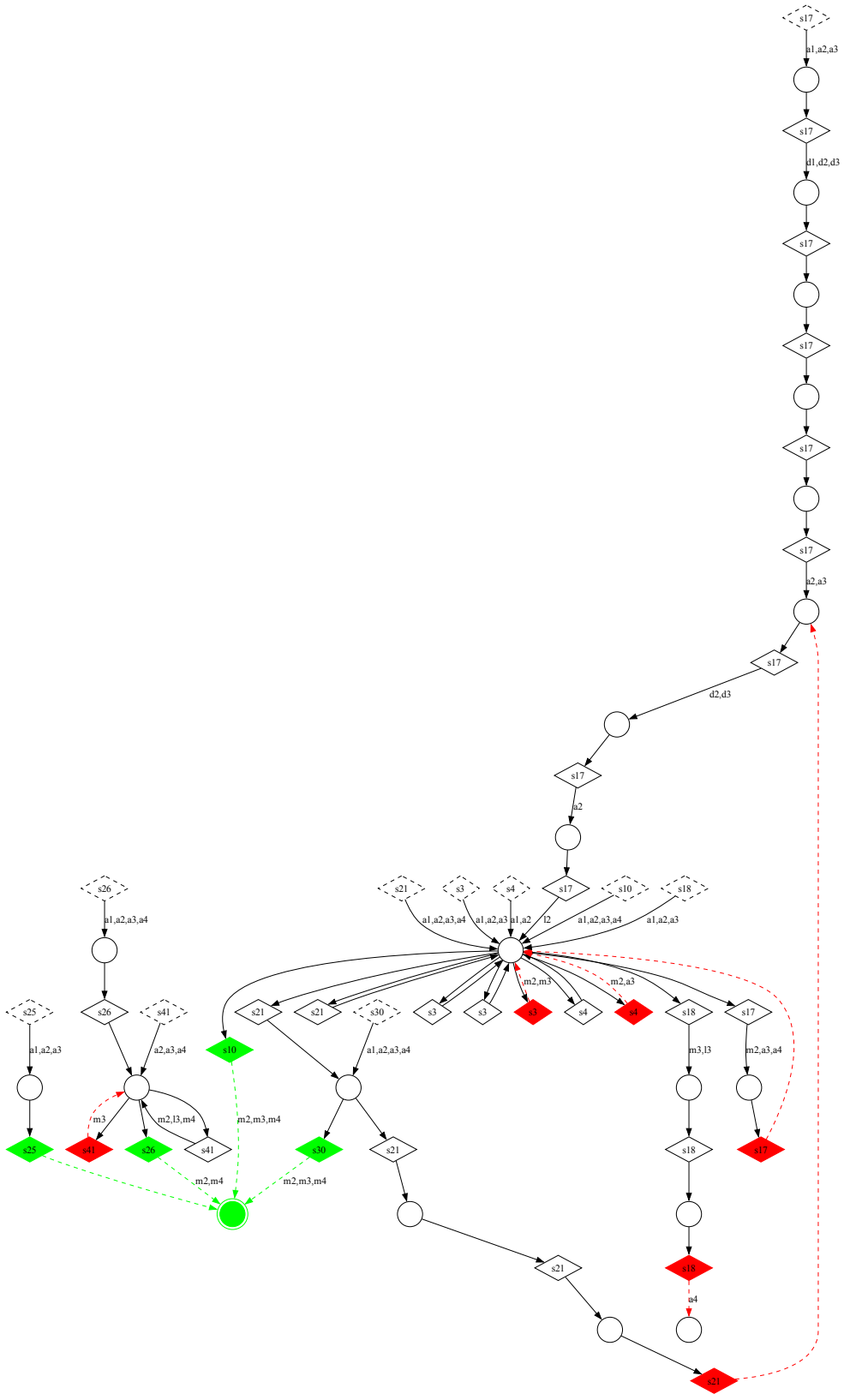

Figure 25: Prompt trajectories for the “changeSection” problem.

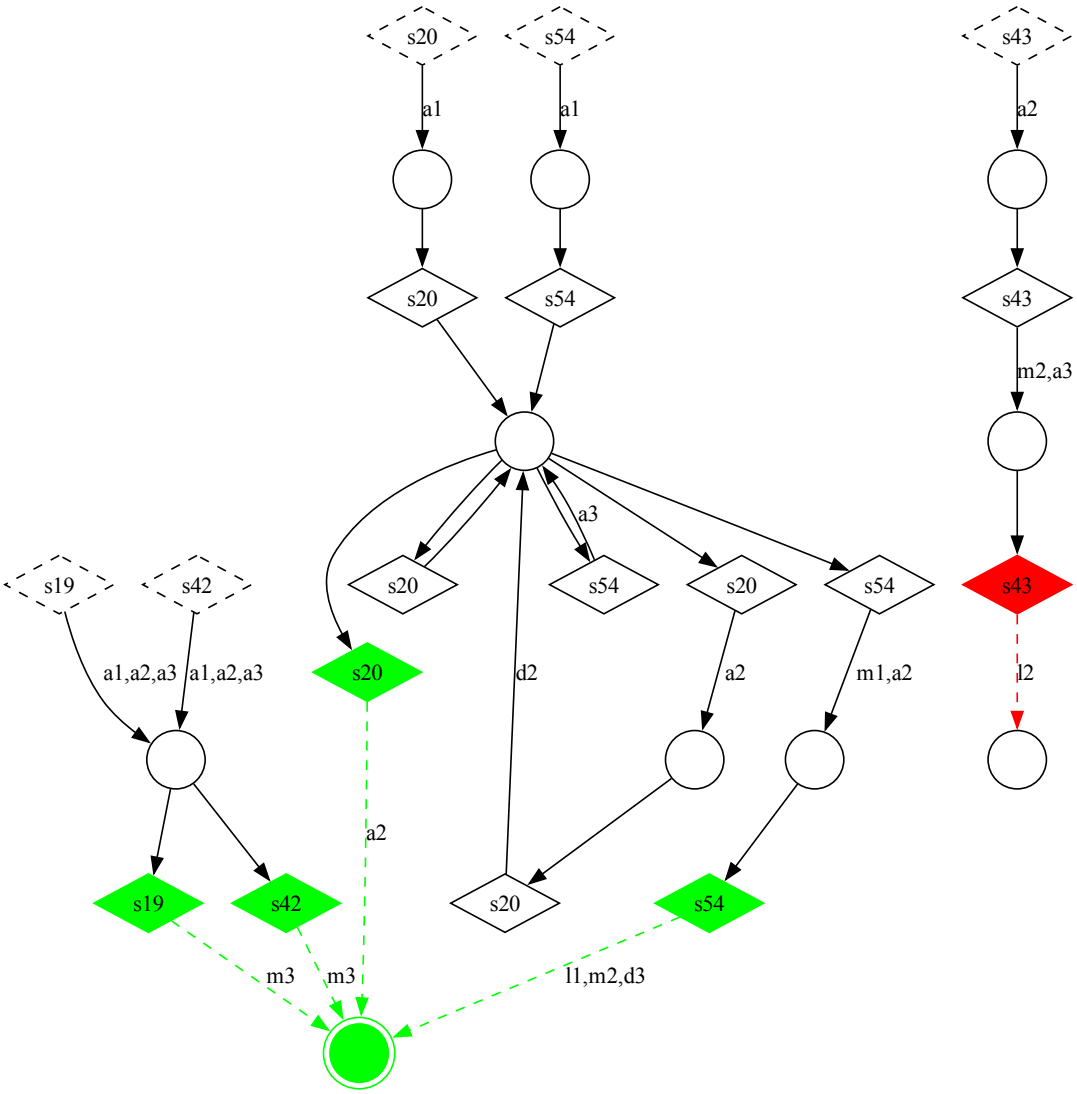

Figure 26: Prompt trajectories for the “combine” problem.

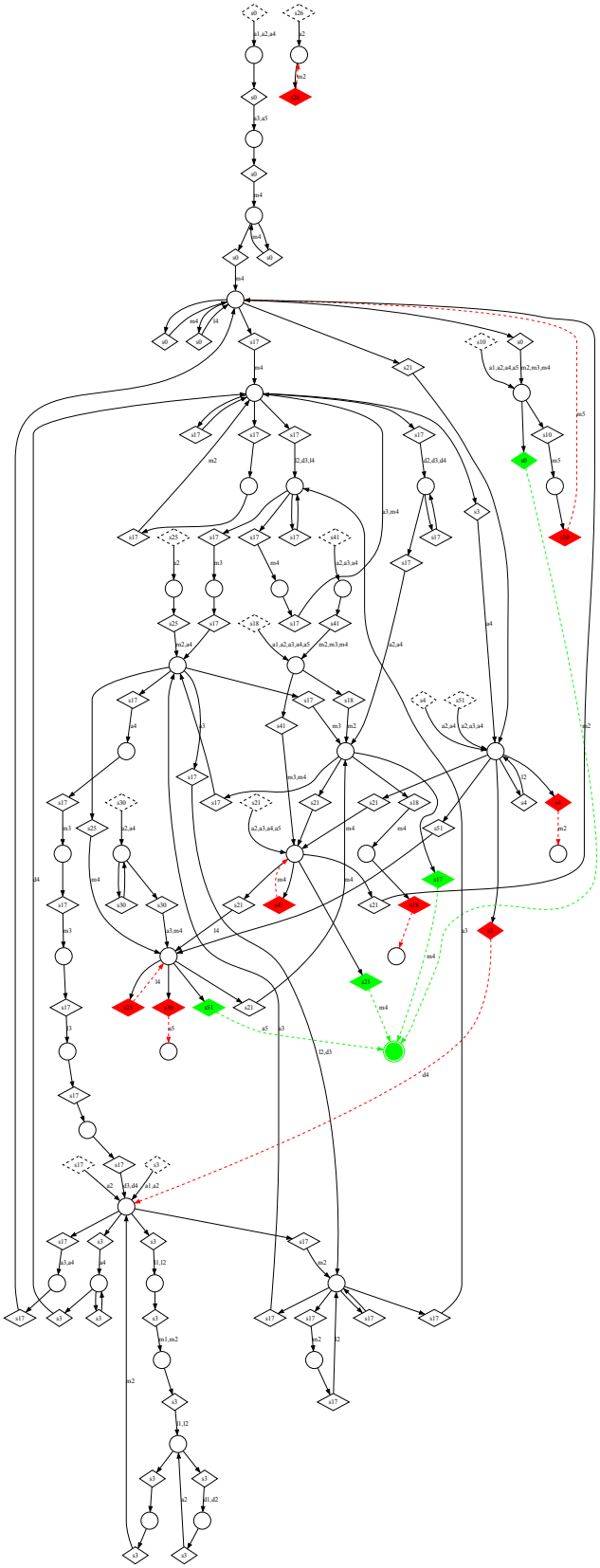

Figure 27: Prompt trajectories for the “convert” problem.

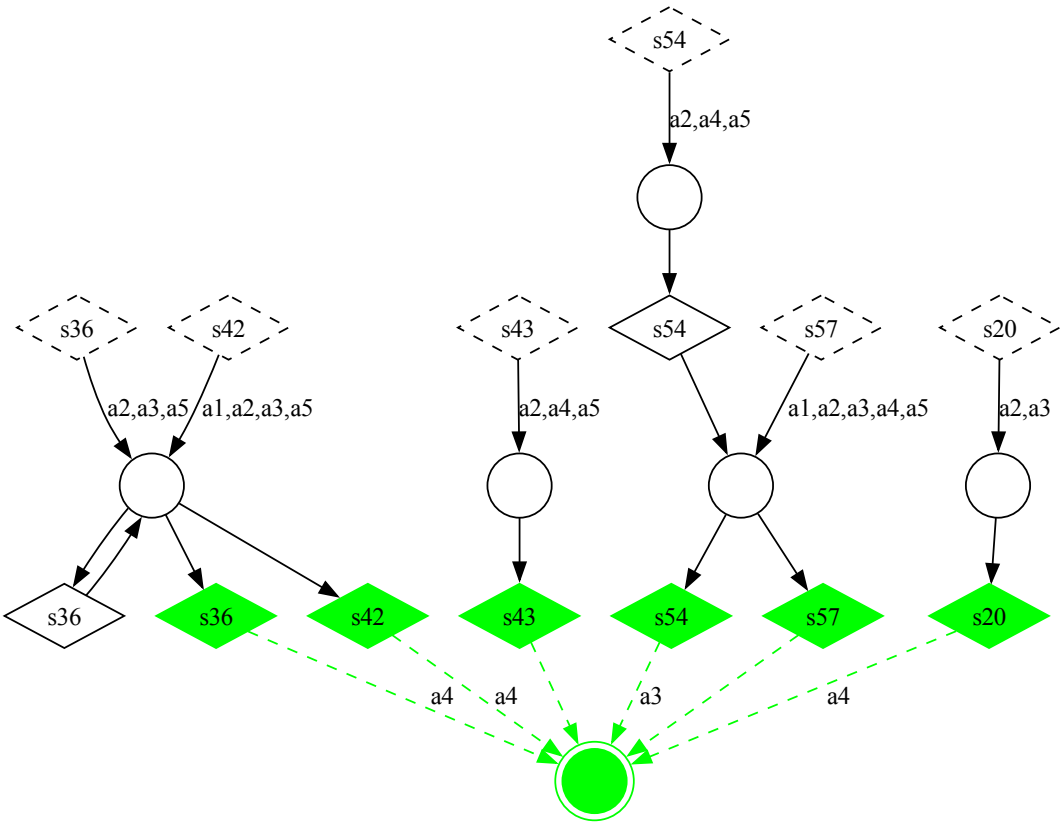

Figure 28: Prompt trajectories for the “create list” problem.

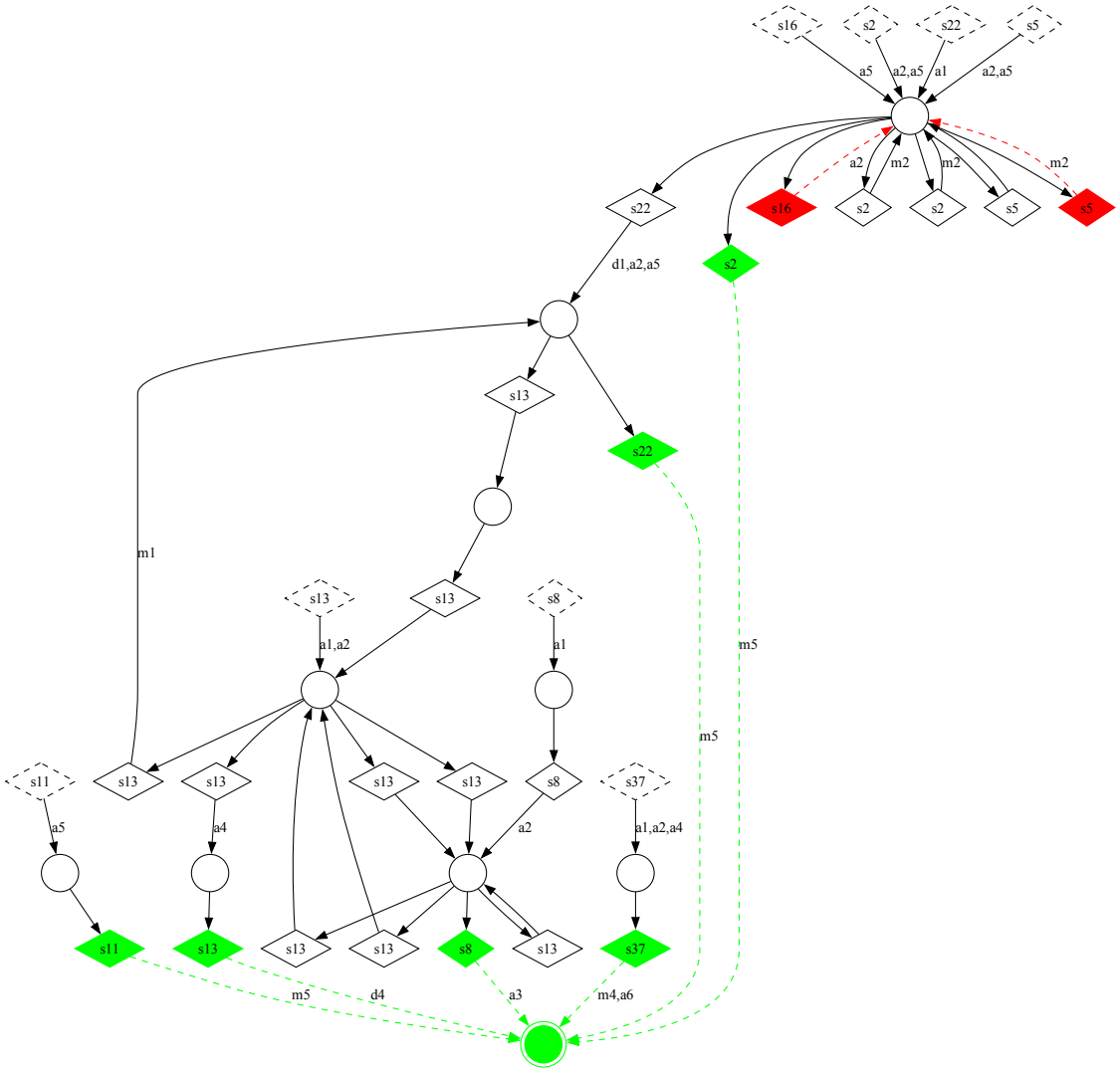

Figure 29: Prompt trajectories for the “fib” problem.

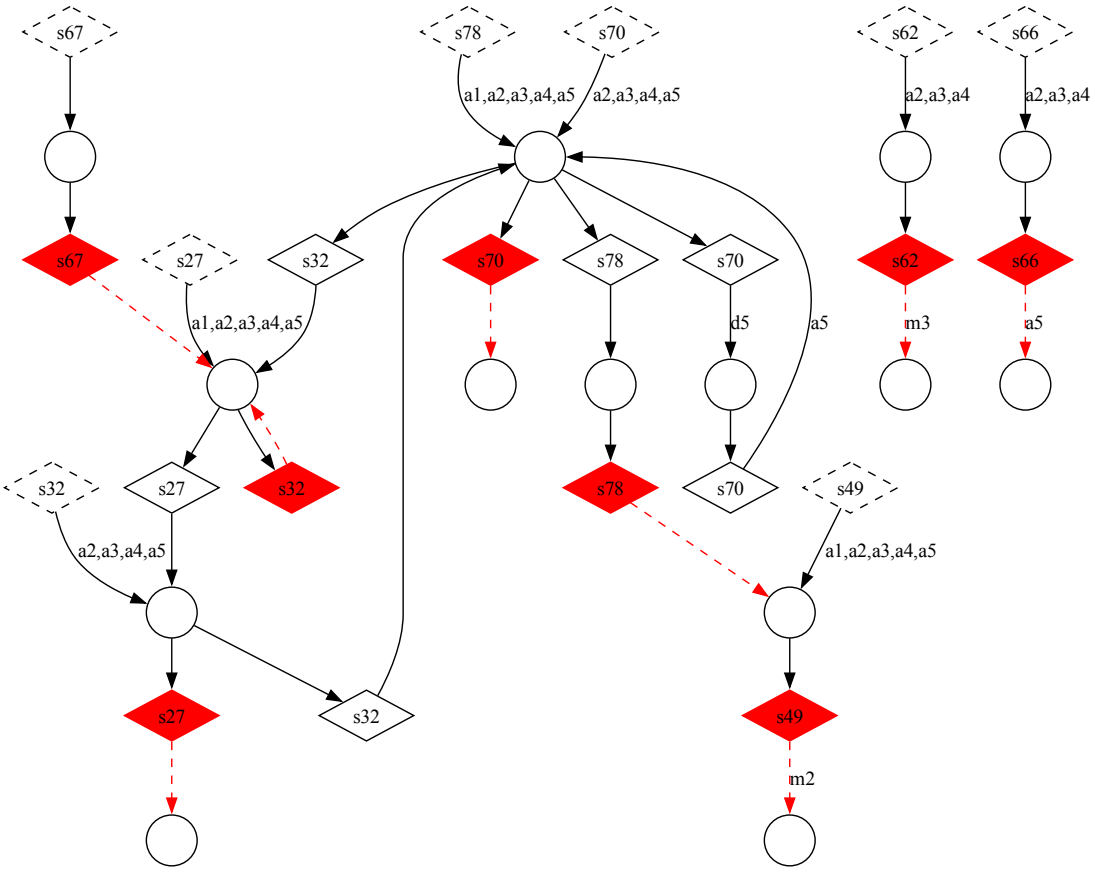

Figure 30: Prompt trajectories for the “findHorizontals” problem.

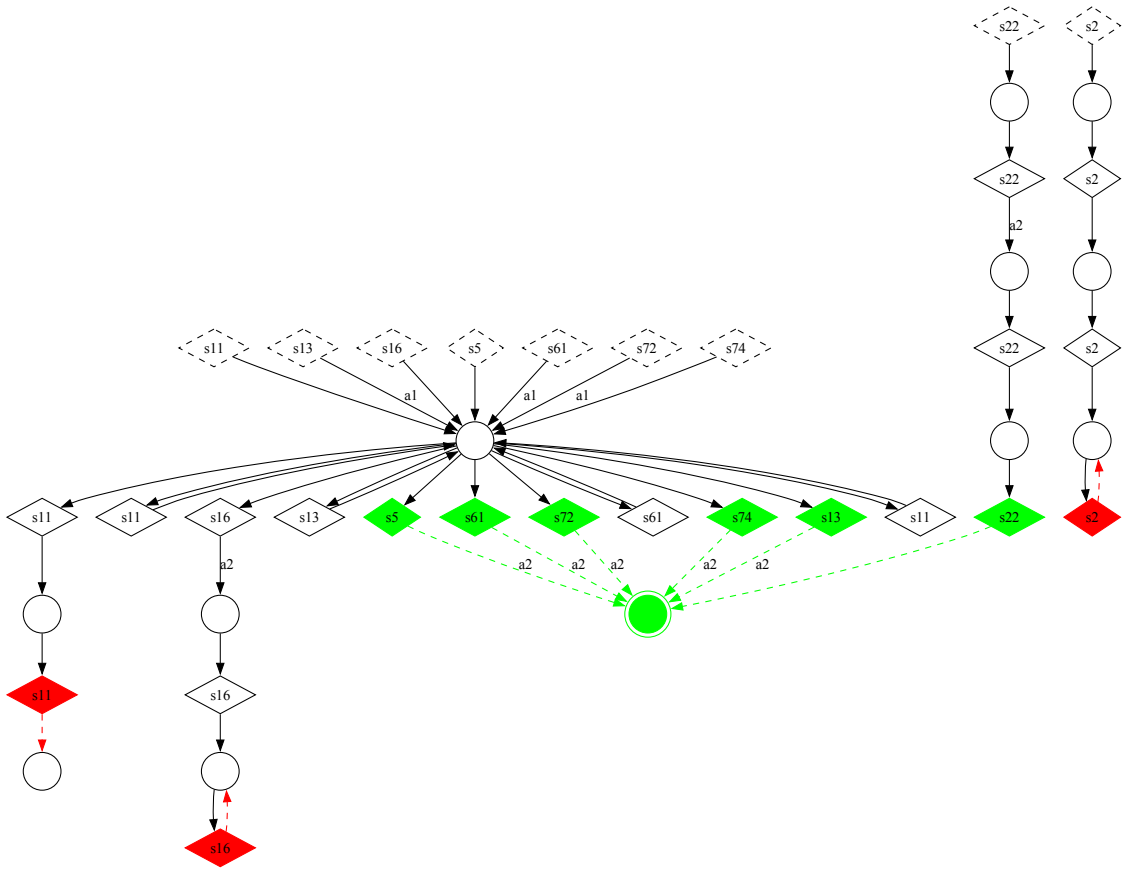

Figure 31: Prompt trajectories for the “find multiples” problem.

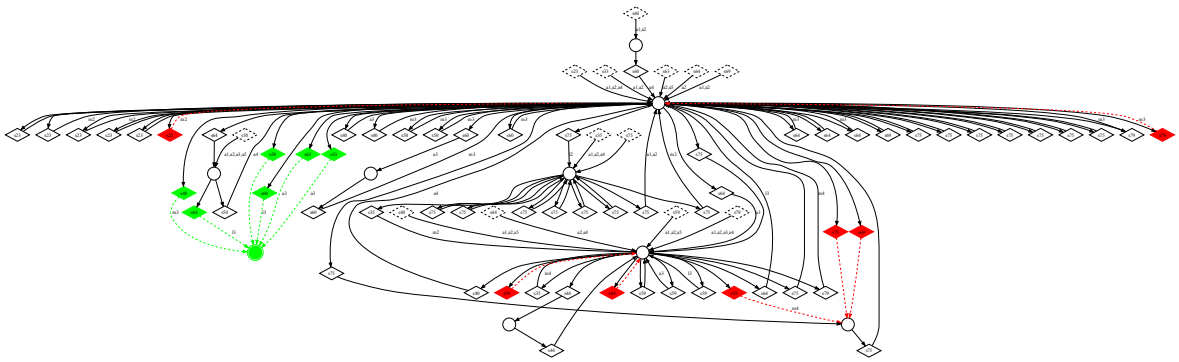

Figure 32: Prompt trajectories for the “generateCardDeck” problem.

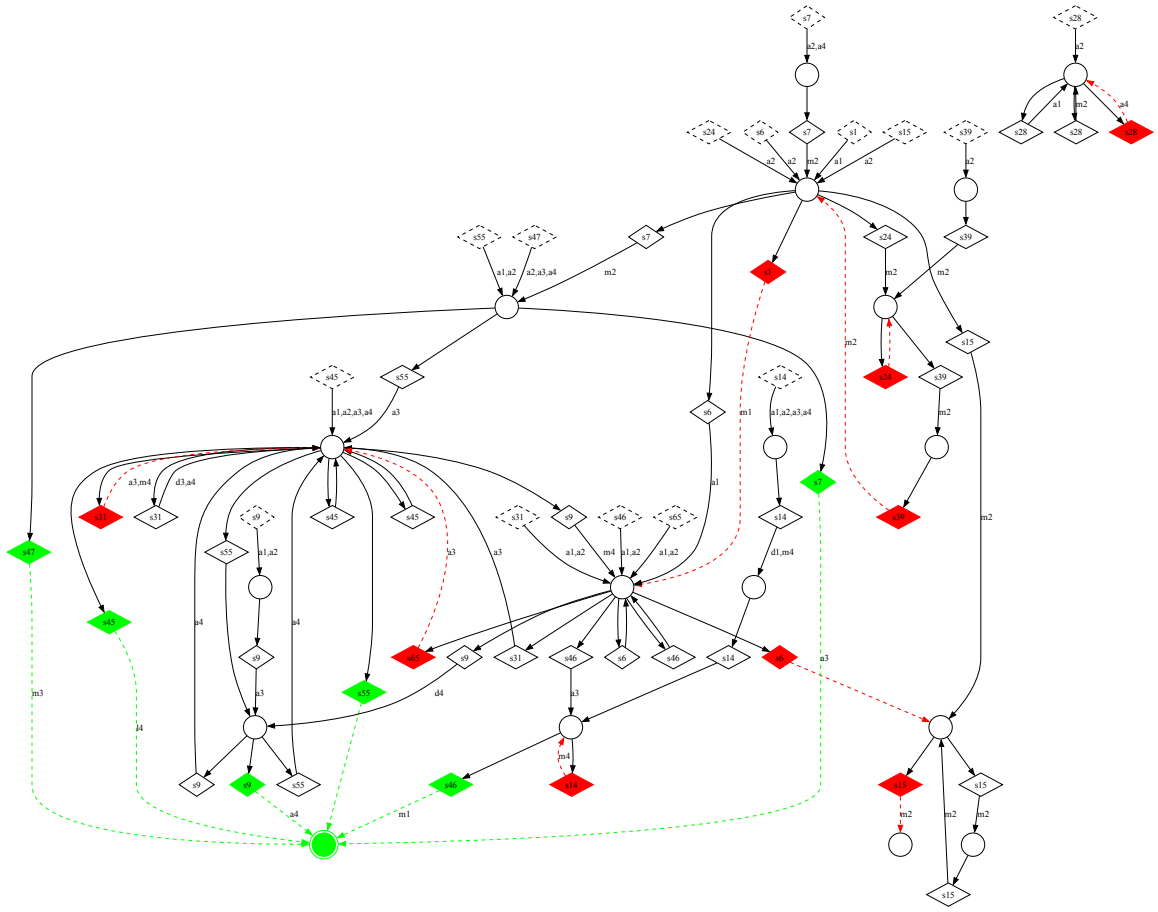

Figure 33: Prompt trajectories for the “getSeason” problem.

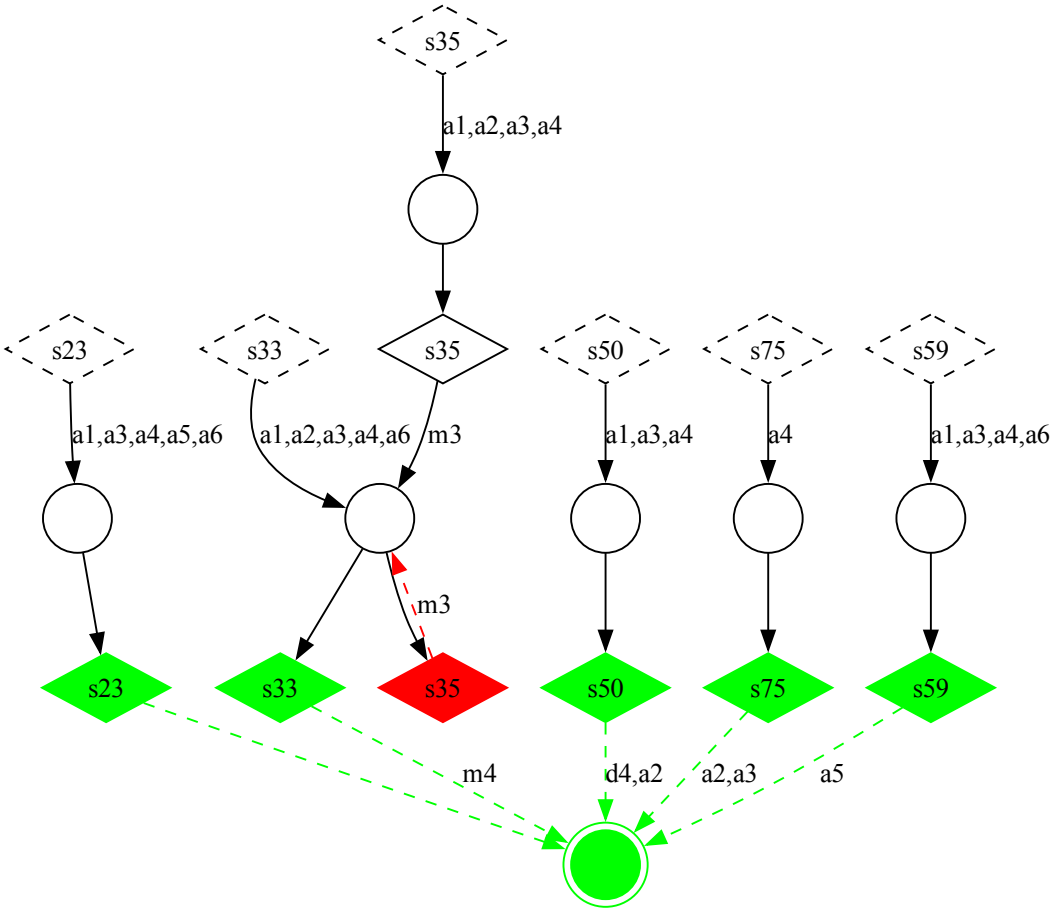

Figure 34: Prompt trajectories for the “increaseScore” problem.

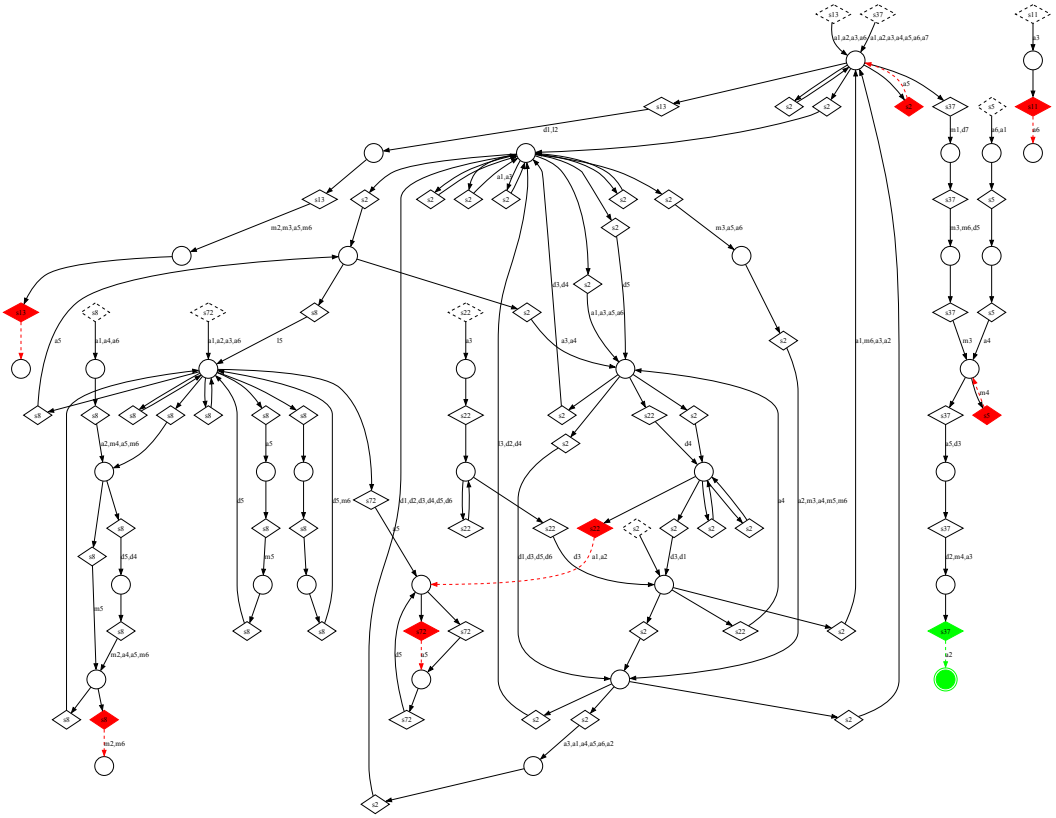

Figure 35: Prompt trajectories for the “laugh” problem.

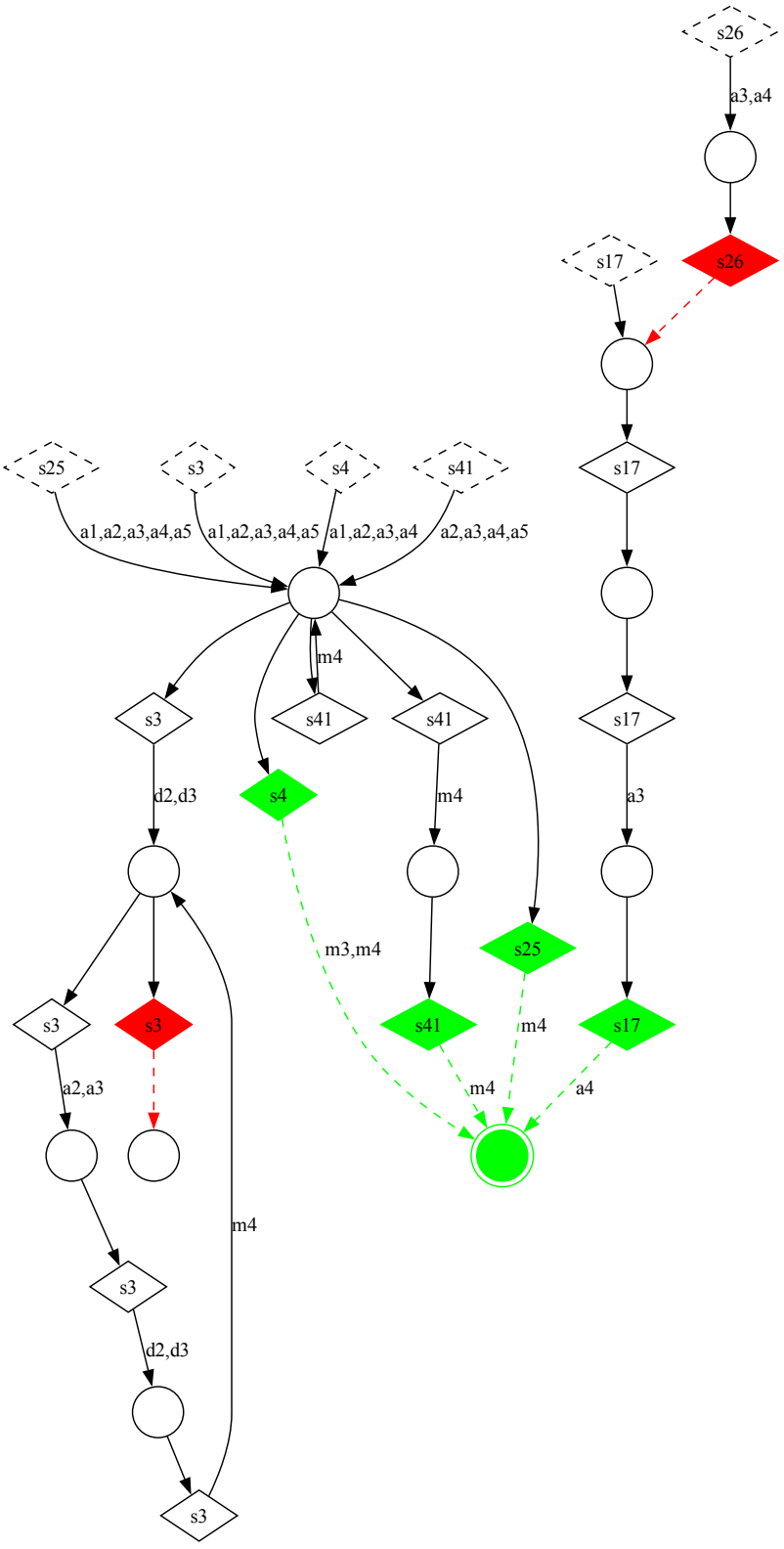

Figure 36: Prompt trajectories for the “pattern” problem.

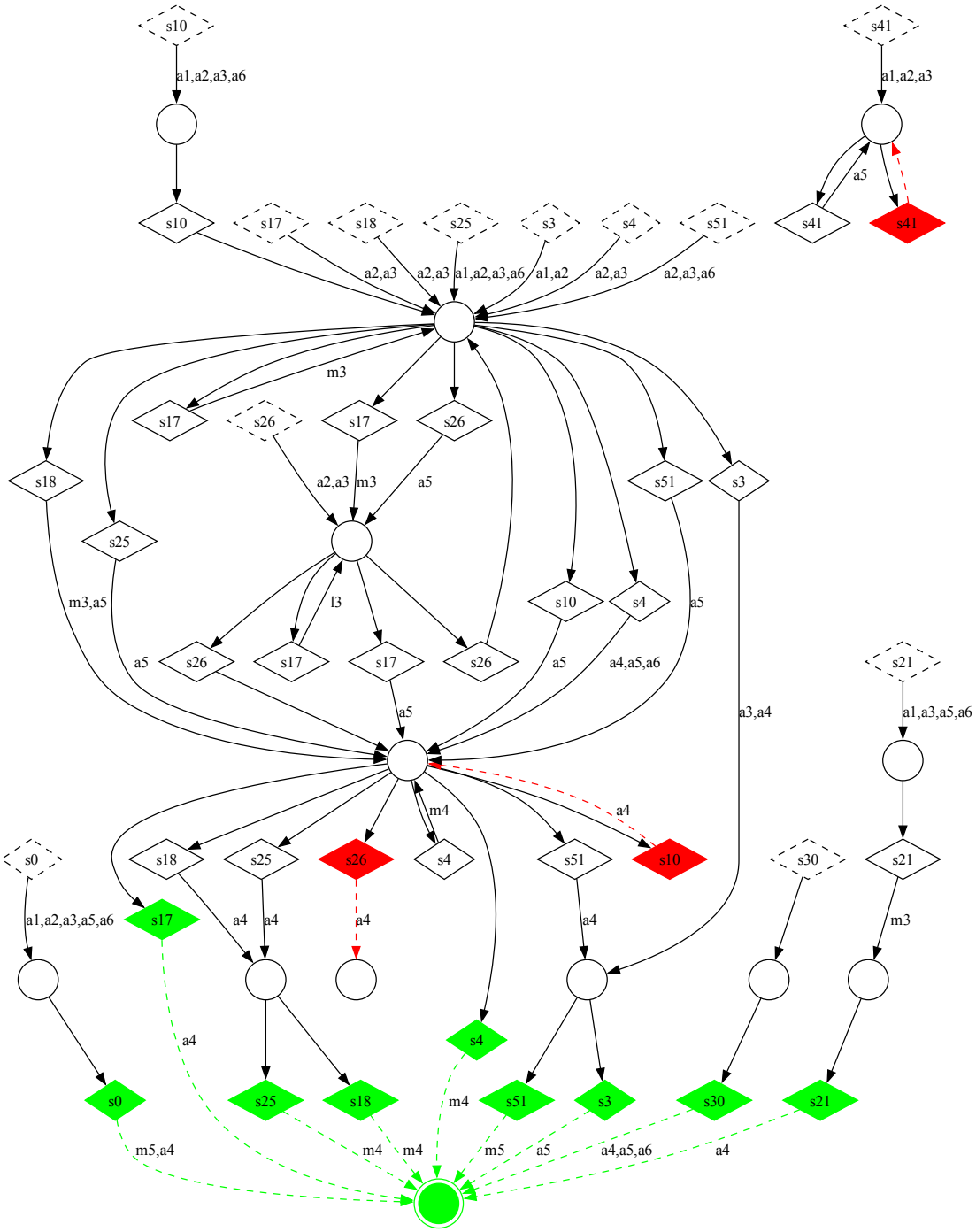

Figure 37: Prompt trajectories for the “percentWin” problem.

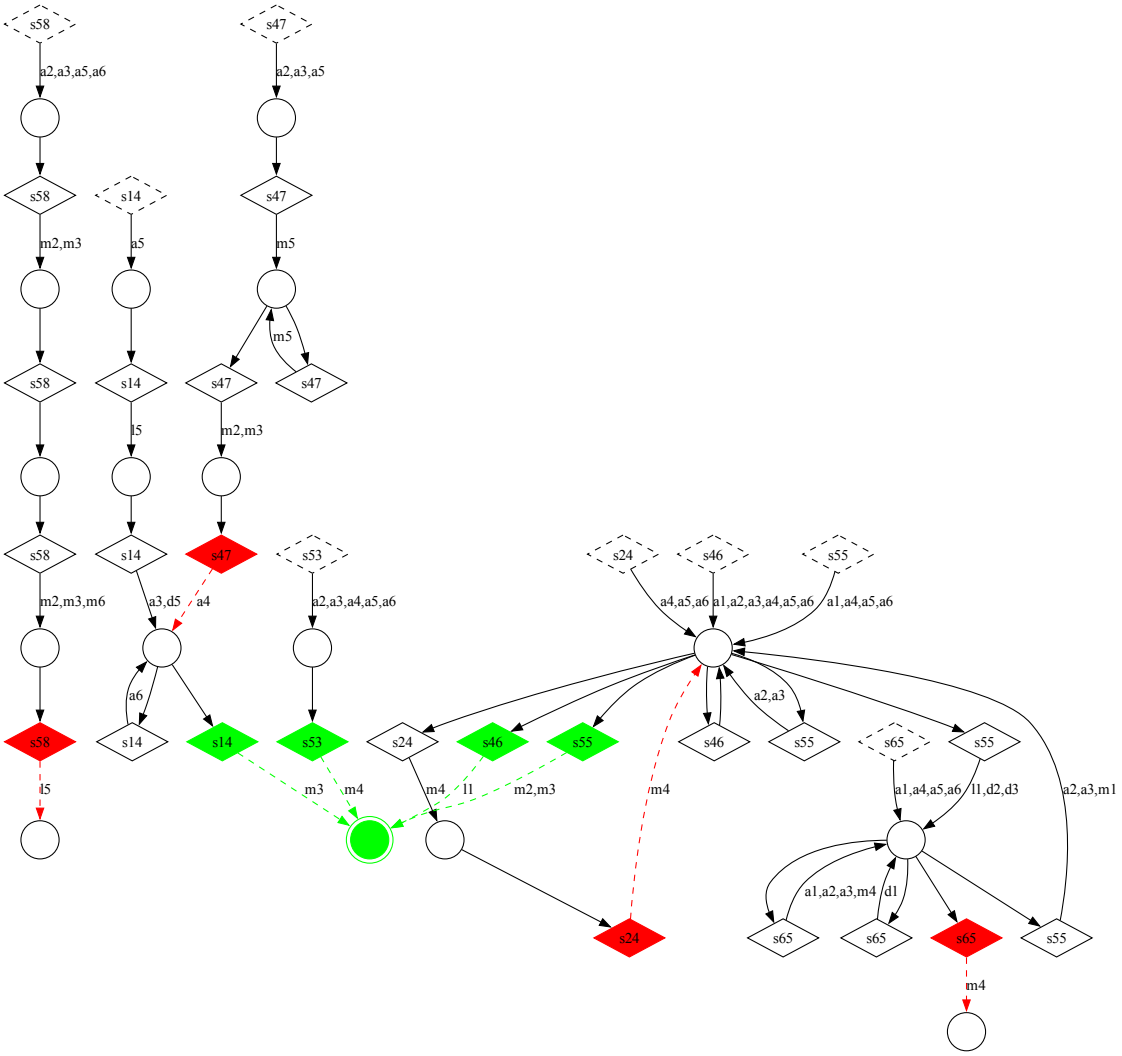

Figure 38: Prompt trajectories for the “planets mass” problem.

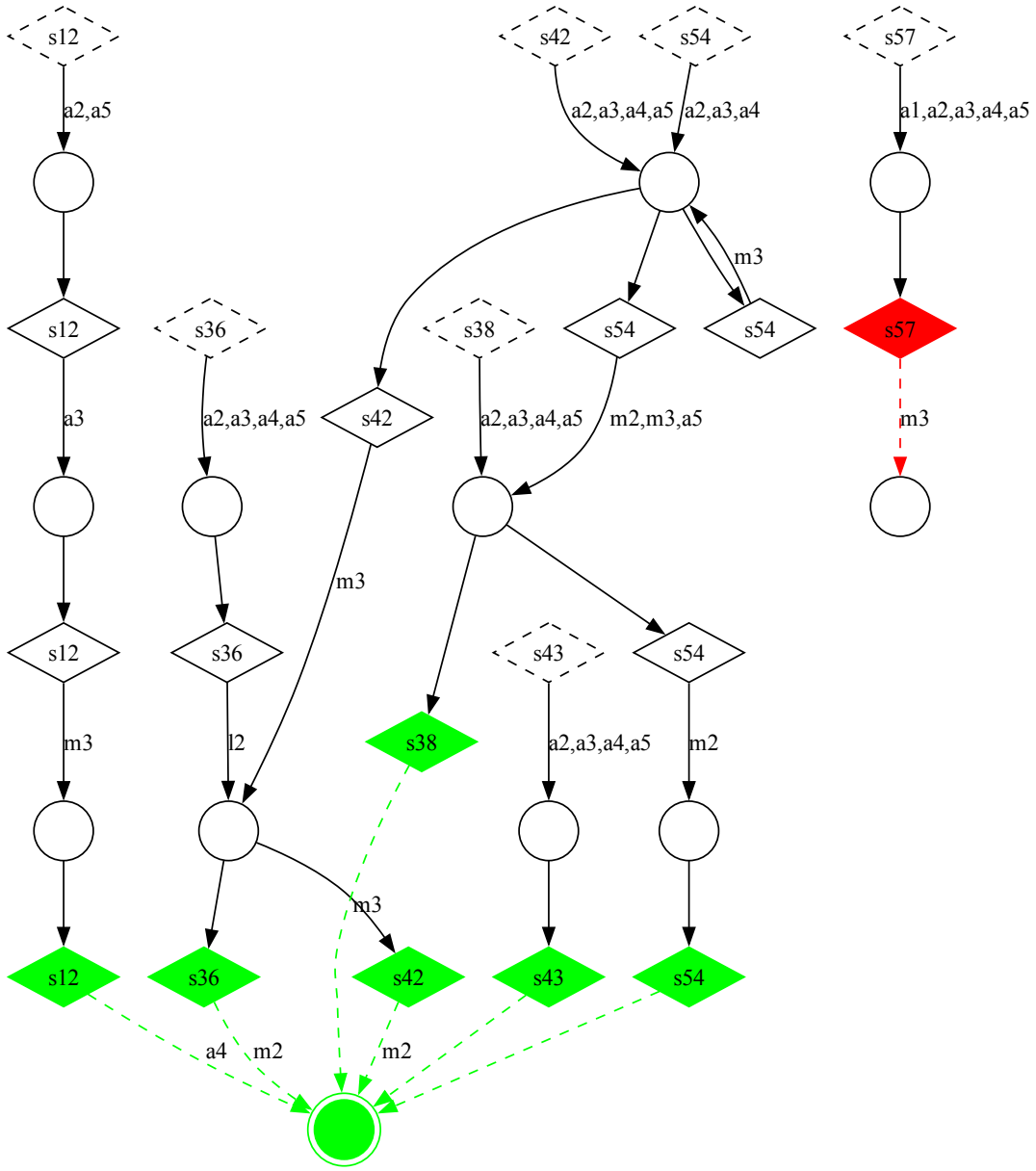

Figure 39: Prompt trajectories for the “print time” problem.

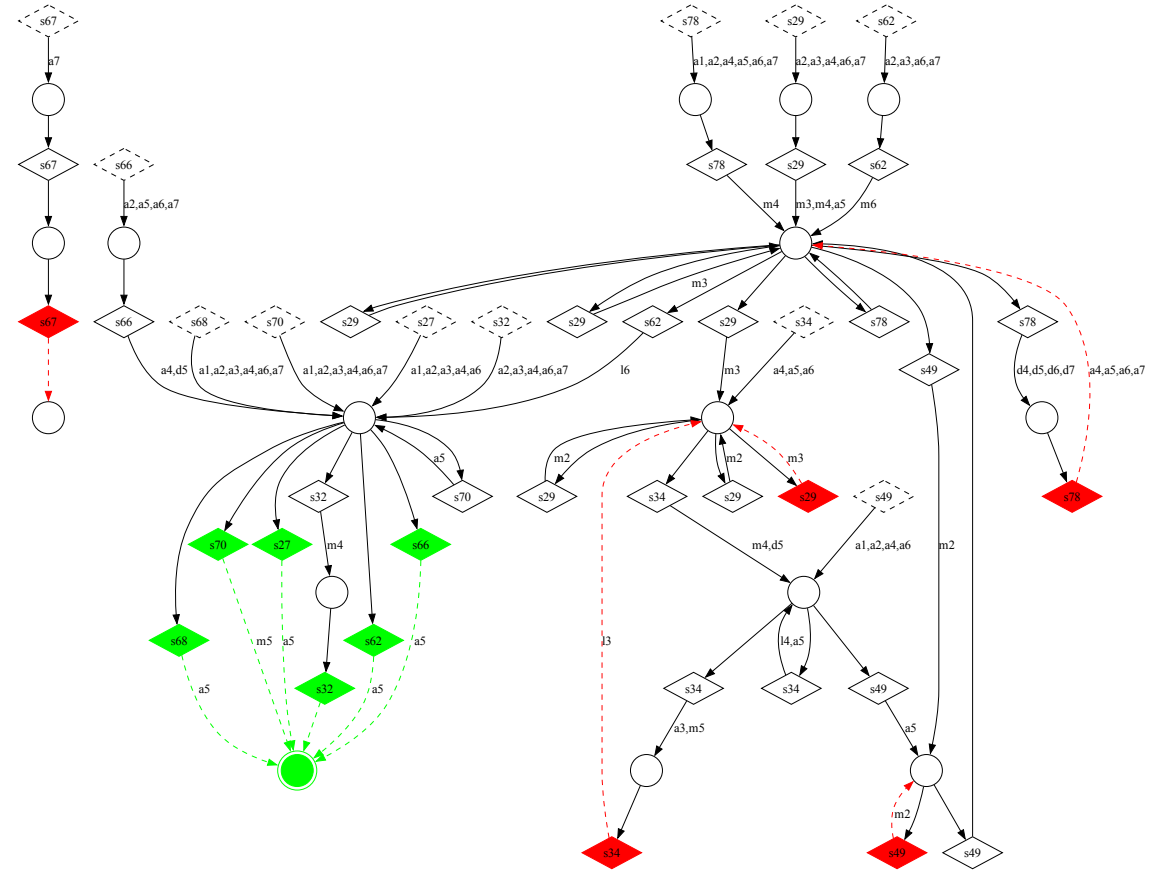

Figure 40: Prompt trajectories for the “readingIceCream” problem.

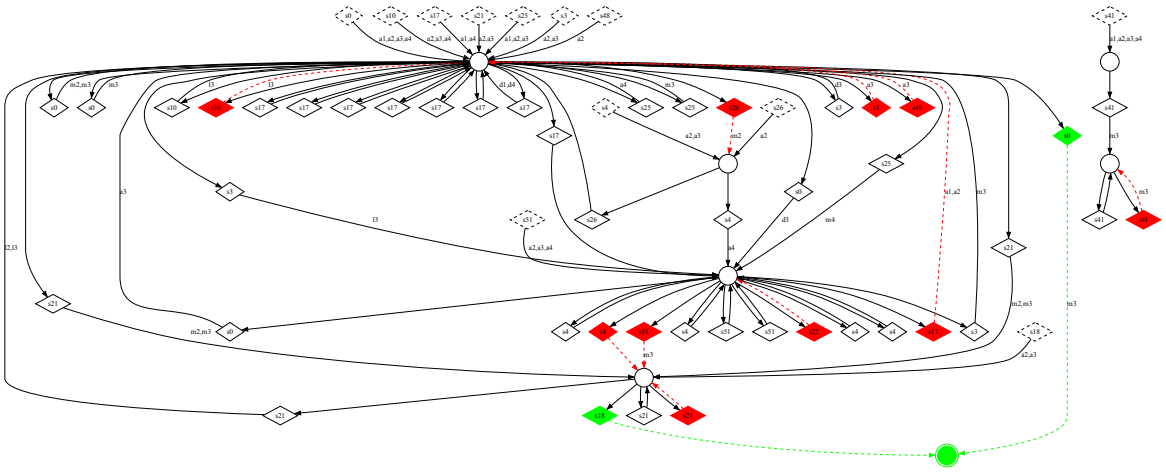

Figure 41: Prompt trajectories for the “remove odd” problem.

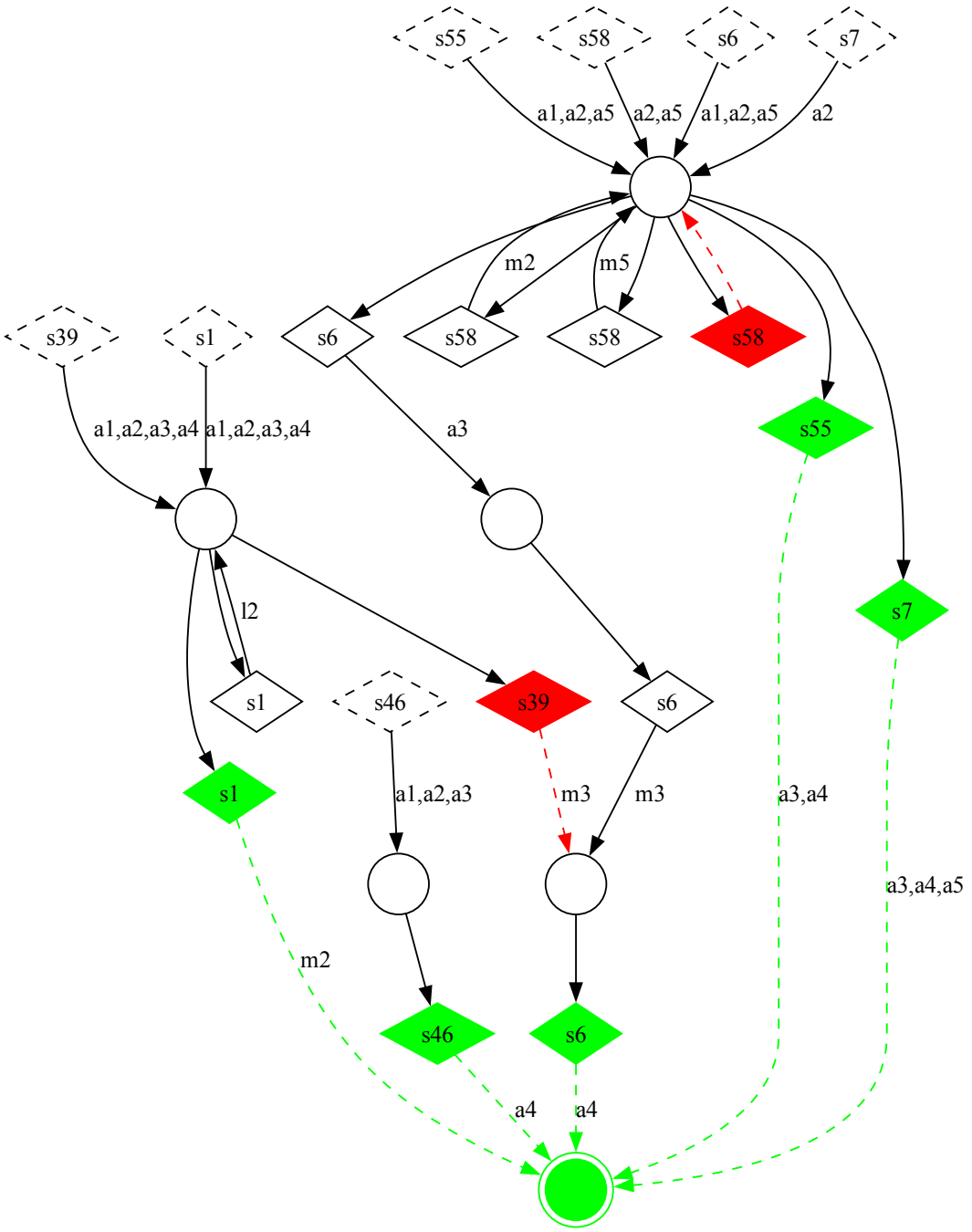

Figure 42: Prompt trajectories for the “reverseWords” problem.

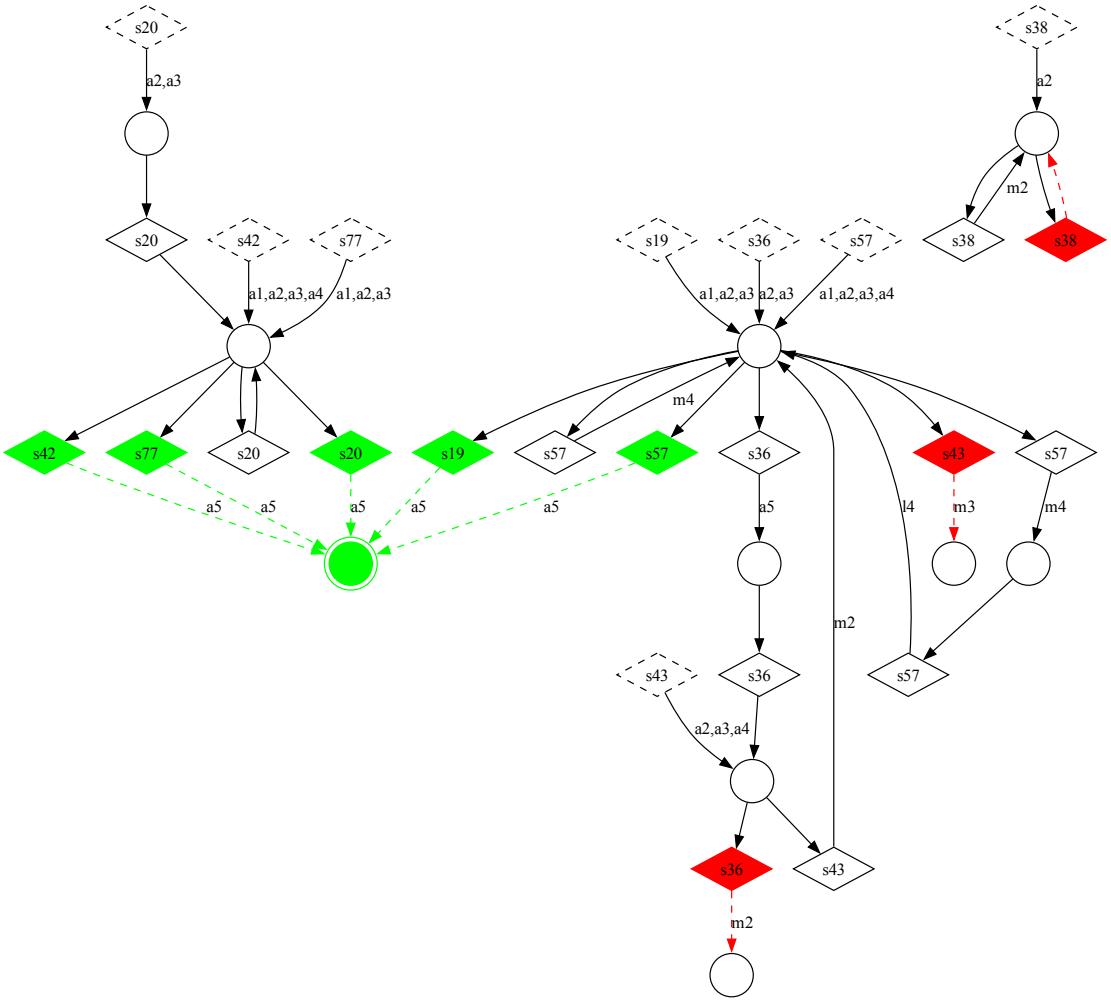

Figure 43: Prompt trajectories for the “set chars” problem.

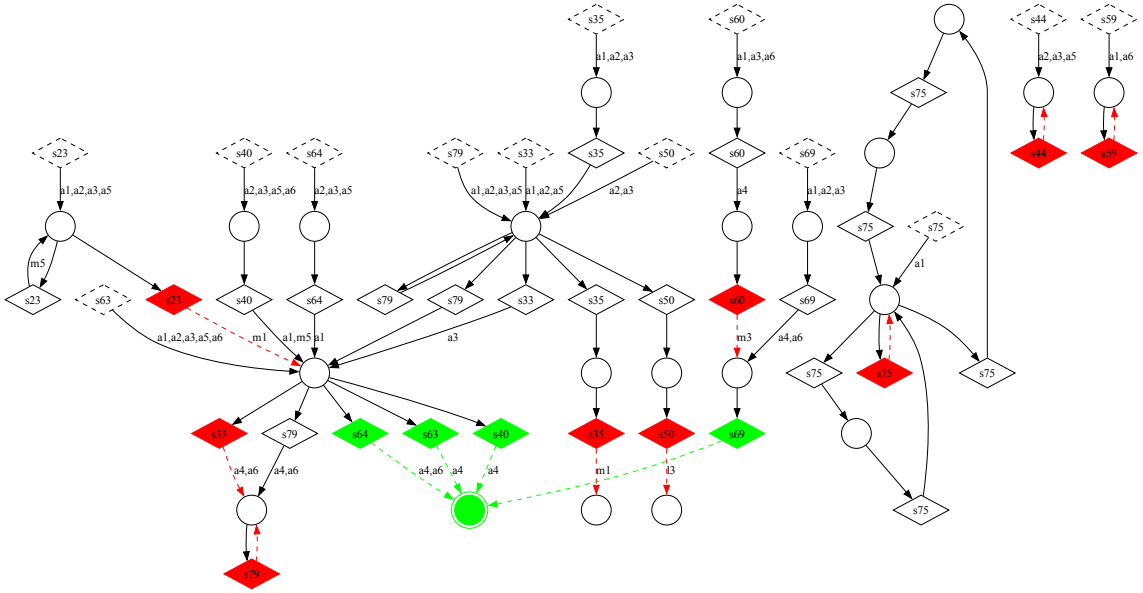

Figure 44: Prompt trajectories for the “sortBySuccessRate” problem.

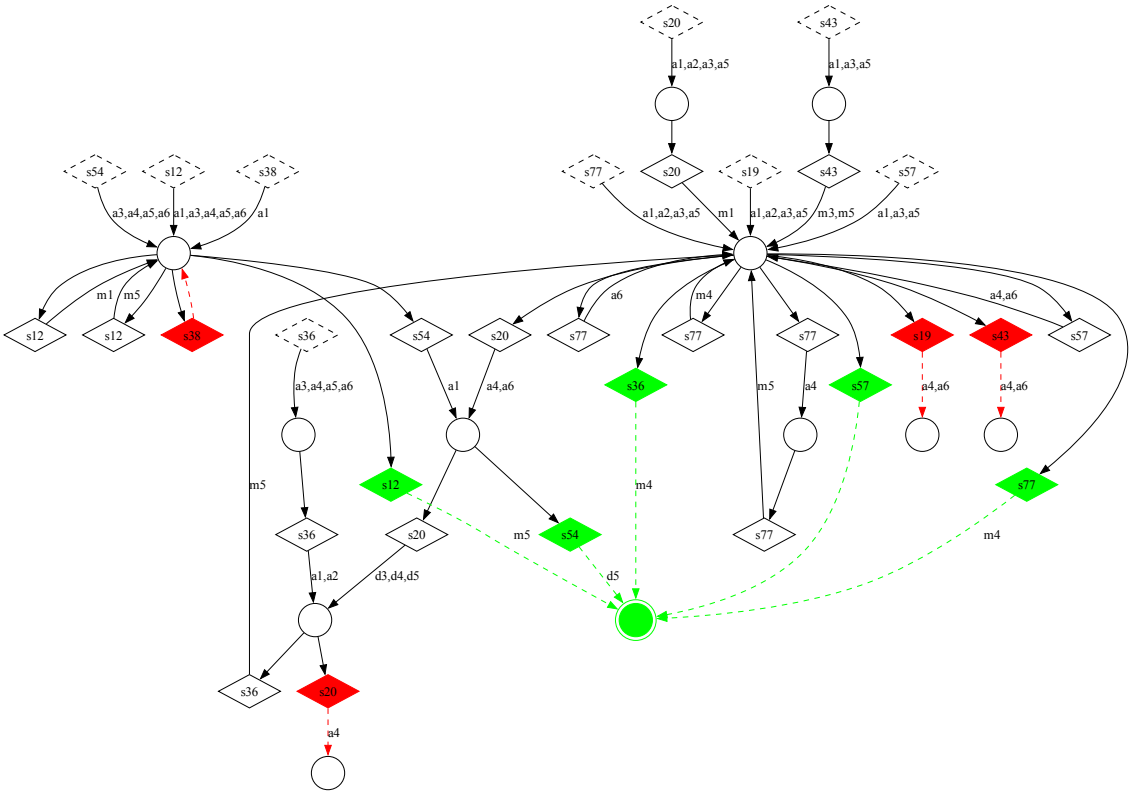

Figure 45: Prompt trajectories for the “sort physicists” problem.

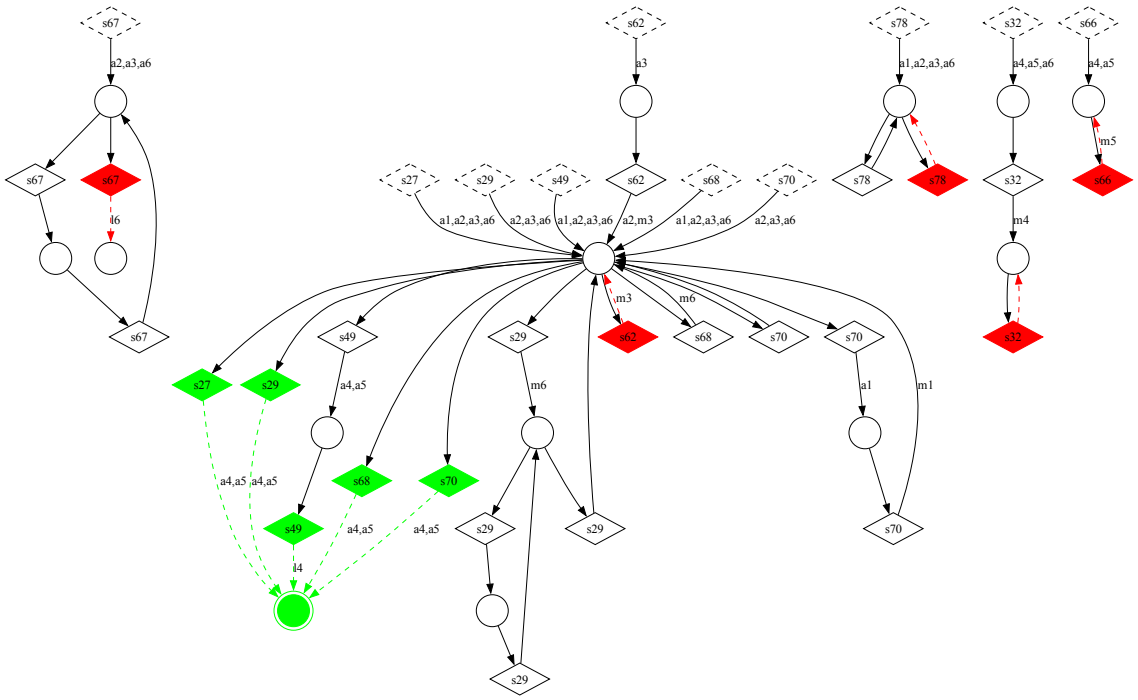

Figure 46: Prompt trajectories for the “sortedBooks” problem.

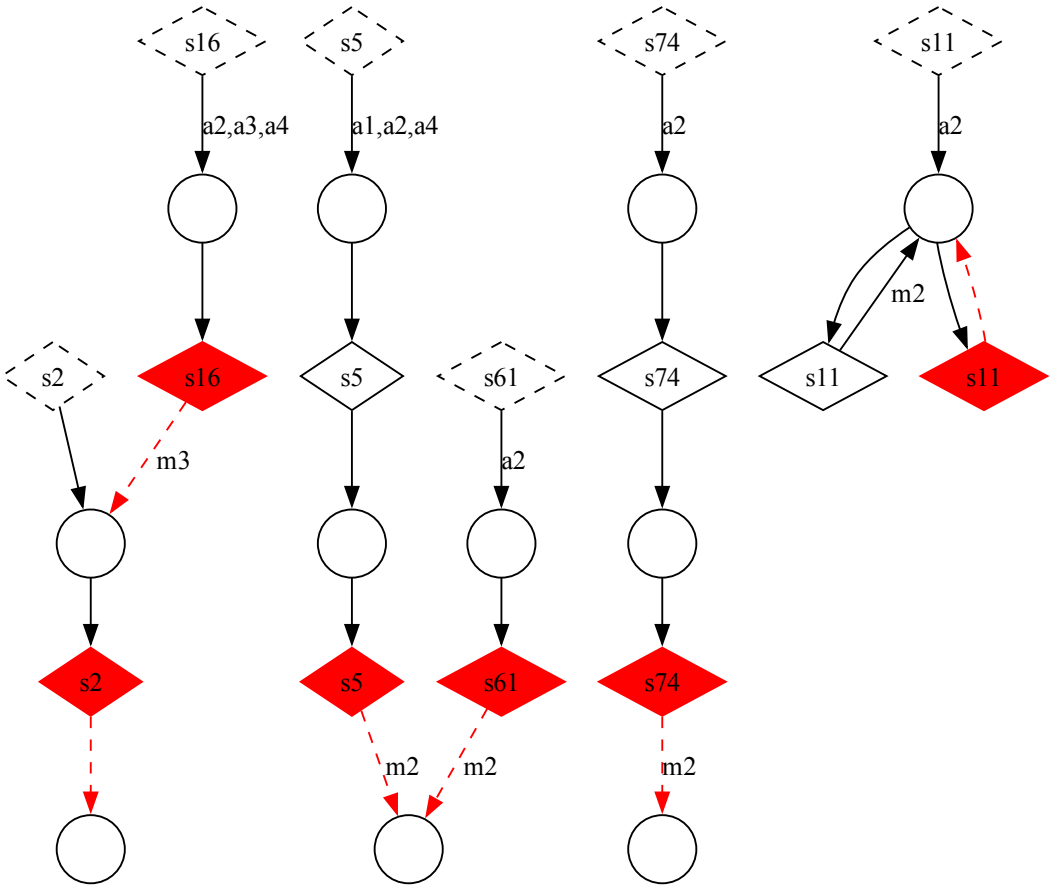

Figure 47: Prompt trajectories for the “student grades” problem.

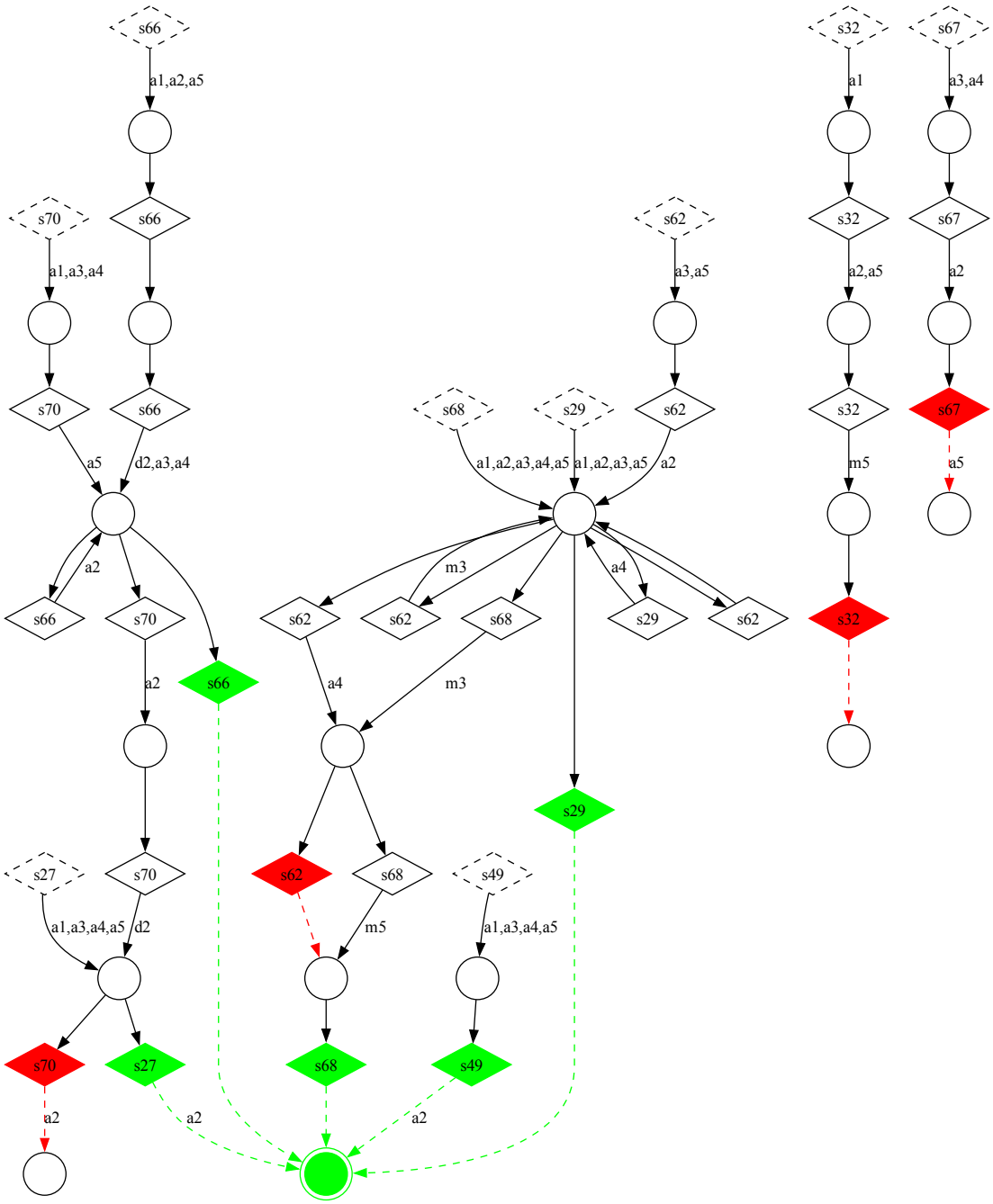

Figure 48: Prompt trajectories for the “subtract add” problem.

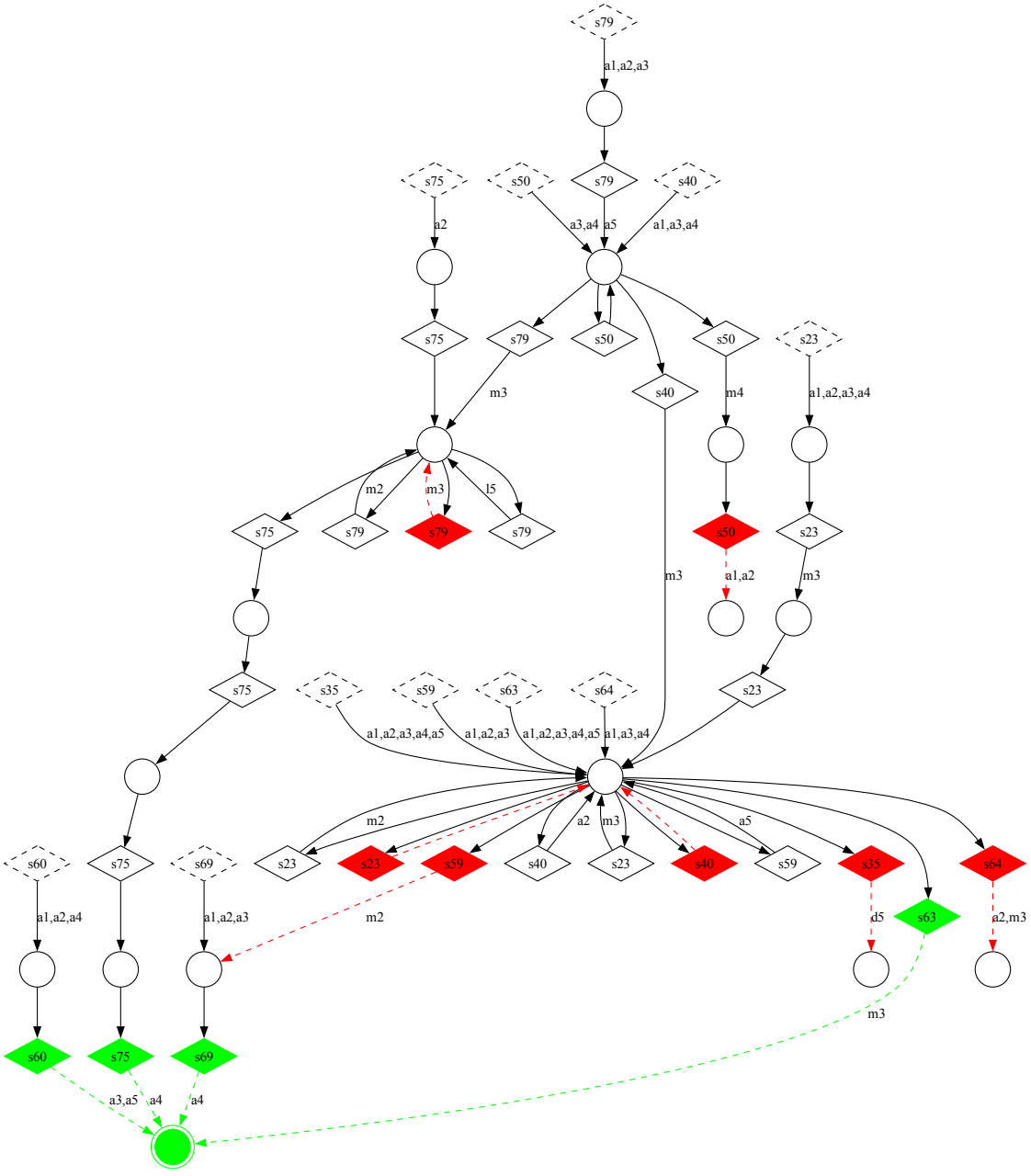

Figure 49: Prompt trajectories for the “times with” problem.

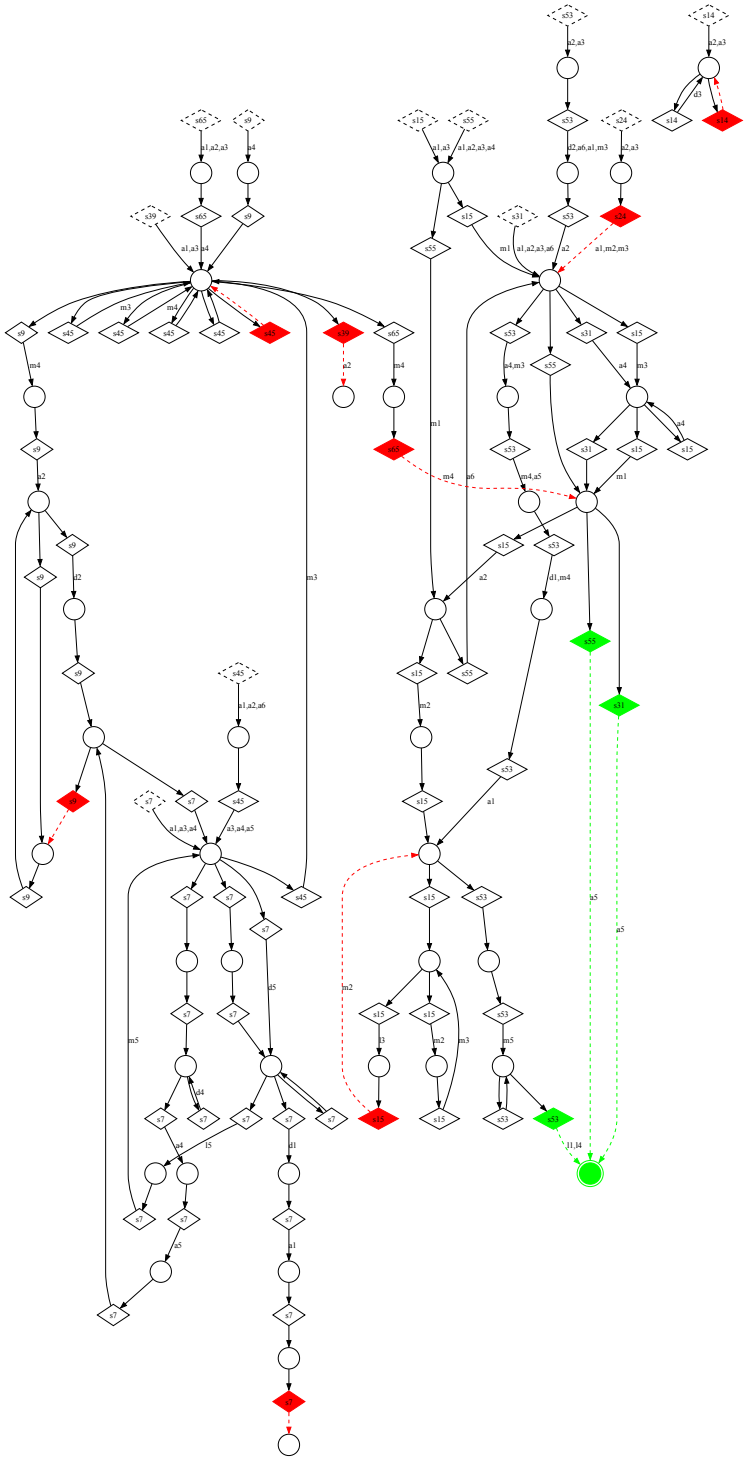

Figure 50: Prompt trajectories for the “topScores” problem.

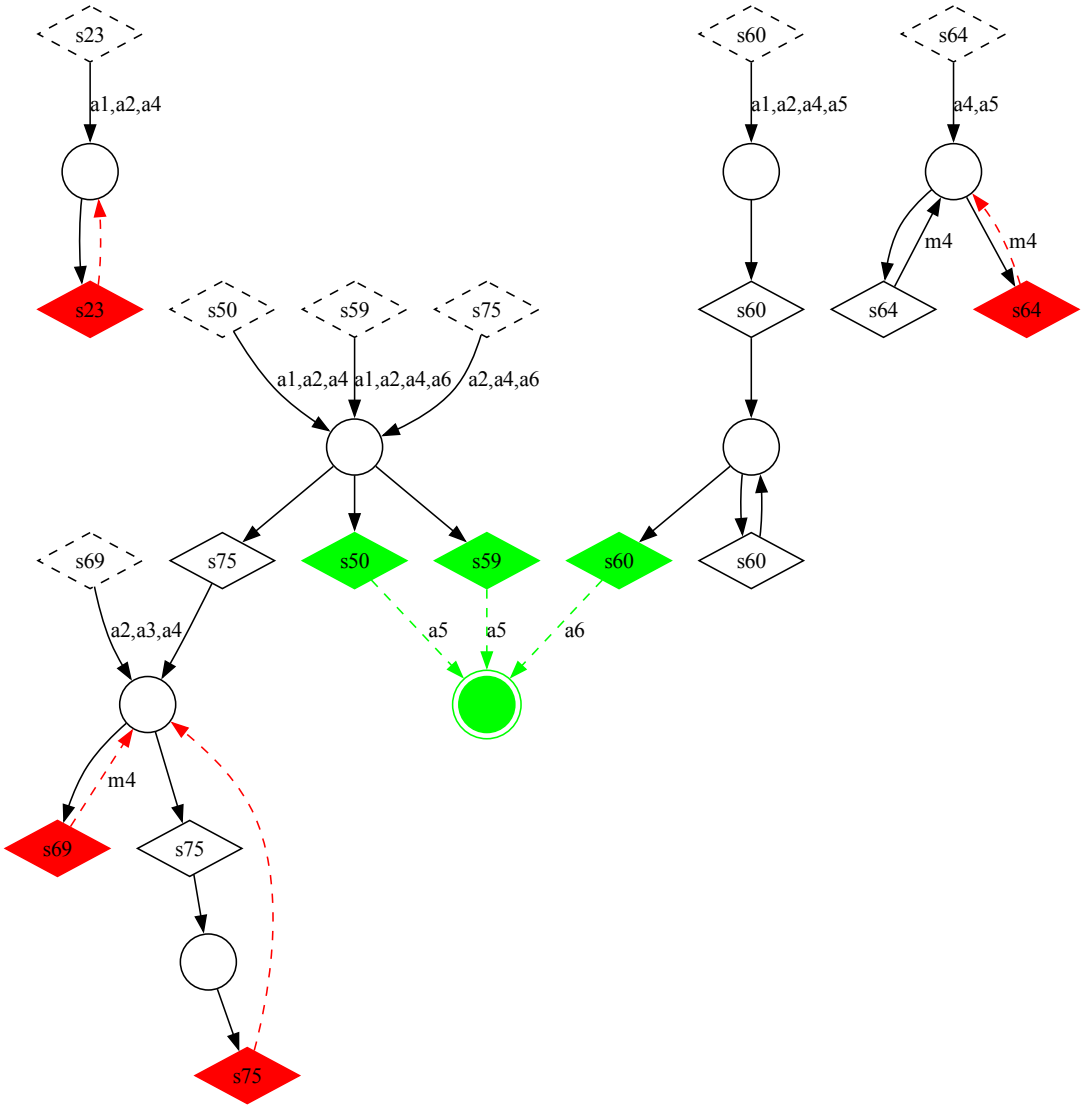

Figure 51: Prompt trajectories for the “translate” problem.

G Checklist

A2 Potential Risks: Did you discuss any potential risks of your work? [Yes/No/NA] **No**

A2 Elaboration: For yes, provide a section number. For no, justify why not. **The risks of this work lie in the original dataset creation and collection, as described by (Nguyen et al., 2024) and (Babe et al., 2024).**

B Use Or Create Scientific Artifacts: Did you use or create scientific artifacts? [Yes/No] **Yes**

B1 Cite Creators Of Artifacts: Did you cite the creators of artifacts you used? [Yes/No/NA] **Yes**

B1 Elaboration: For yes, provide a section number. For no, justify why not. **§3**

B2 Discuss The License For Artifacts: Did you discuss the license or terms for use and/or distribution of any artifacts? [Yes/No/NA] **Yes**

B2 Elaboration: For yes, provide a section number. For no, justify why not. **Appendix A**

B3 Artifact Use Consistent With Intended Use: Did you discuss if your use of existing artifact(s) was consistent with their intended use, provided that it was specified? For the artifacts you create, do you specify intended use and whether that is compatible with the original access conditions? [Yes/No/NA] **Yes**

B3 Elaboration: For yes, provide a section number. For no, justify why not. **Ethics Statement**

B4 Data Contains Personally Identifying Info Or Offensive Content: Did you discuss the steps taken to check whether the data that was collected/used contains any information that names or uniquely identifies individual people or offensive content, and the steps taken to protect/anonymize it? [Yes/No/NA] **NA**

B4 Elaboration: For yes, provide a section number. For no, justify why not.

B5 Documentation Of Artifacts: Did you provide documentation of the artifacts, e.g., coverage of domains, languages, and linguistic phenomena, demographic groups represented, etc.? [Yes/No/NA] **No**

B5 Elaboration: For yes, provide a section number. For no, justify why not. **This paper analyzes an existing dataset, which documents the artifact per ACL guidelines.**

B6 Statistics For Data: Did you report relevant statistics like the number of examples, details of train/test/dev splits, etc. for the data that you used/created? [Yes/No/NA] **Yes**

B6 Elaboration: For yes, provide a section number. For no, justify why not. **§3**

C Computational Experiments: Did you run computational experiments? [Yes/No/NA] **Yes**

C1 Model Size And Budget: Did you report the number of parameters in the models used, the total computational budget (e.g., GPU hours), and computing infrastructure used? [Yes/No/NA] **Yes**

C1 Elaboration: For yes, provide a section number. For no, justify why not. **Appendix C**

C2 Experimental Setup And Hyperparameters: Did you discuss the experimental setup, including hyperparameter search and best-found hyperparameter values? [Yes/No/NA] **Yes**

C2 Elaboration: For yes, provide a section number. For no, justify why not. **Appendix E.3**

C3 Descriptive Statistics: Did you report descriptive statistics about your results (e.g., error bars around results, summary statistics from sets of experiments), and is it transparent whether you are reporting the max, mean, etc. or just a single run? [Yes/No/NA] **Yes**

C3 Elaboration: For yes, provide a section number. For no, justify why not. **§4.1.4 and Appendix E.4**

C4 Parameters For Packages: If you used existing packages (e.g., for preprocessing, for normalization, or for evaluation, such as NLTK, SpaCy, ROUGE, etc.), did you report the implementation, model, and parameter settings used? [Yes/No/NA] **Yes**

C4 Elaboration: For yes, provide a section number. For no, justify why not. **Appendix D**

D Human Subjects Including Annotators: Did you use human annotators (e.g., crowdworkers) or research with human subjects? [Yes/No/NA] **No**

D1 Instructions Given To Participants: Did you report the full text of instructions given to participants, including e.g., screenshots, disclaimers of any risks to participants or annotators, etc.? [Yes/No/NA] **NA**

D1 Elaboration: For yes, provide a section number. For no, justify why not.

D2 Recruitment And Payment: Did you report information about how you recruited (e.g., crowdsourcing platform, students) and paid participants, and discuss if such payment is adequate given the participants' demographic (e.g., country of residence)? [Yes/No/NA] **NA**

D2 Elaboration: For yes, provide a section number. For no, justify why not.

D3 Data Consent: Did you discuss whether and how consent was obtained from people whose data you're using/curating (e.g., did your instructions explain how the data would be used)? [Yes/No/NA] **NA**

D3 Elaboration: For yes, provide a section number. For no, justify why not.

D4 Ethics Review Board Approval: Was the data collection protocol approved (or determined exempt) by an ethics review board? [Yes/No/NA] **NA**

D4 Elaboration: For yes, provide a section number. For no, justify why not.

D5 Characteristics Of Annotators: Did you report the basic demographic and geographic characteristics of the annotator population that is the source of the data? [Yes/No/NA] **NA**

D5 Elaboration: For yes, provide a section number. For no, justify why not.

E Ai Assistants In Research Or Writing: Did you use AI assistants (e.g., ChatGPT, Copilot) in your research, coding, or writing? [Yes/No] **Yes**

E1 Information About Use Of AI Assistants: Did you include information about your use of AI assistants? [Yes/No/NA] **Yes**

E1 Elaboration: For yes, provide a section number. For no, justify why not. **Appendix B**