# Introduction/Literature Review

## Software engineering

## Framing, accountable disciplinary knowledge

## Studying the novice programmer

# Context, Research Questions, and Methods

## Context of the study

The data presented in this case are taken from an ongoing IRB-approved study of undergraduate electrical engineering majors undertaken at a Mid-Atlantic State, a large, public research institution on the east coast. For two semesters, I have followed students taking “Intermediate Programming Concepts for Engineers.” It is the second of a required two-semester course sequence in programming.[[1]](#footnote-1) Students can (and some do) place out of Beginning Programming via AP Computer Science credit, but all Electrical and Computer Engineering students must take Intermediate Programming.

Intermediate Programming has two 75-minute lectures per week and a weekly discussion section led by an undergraduate Teaching Assistant (TA). Typical enrollment is between 60 and 80 students per semester. Like Beginning Programming, Intermediate Programming is taught using the C programming language, and it incorporates multi-week projects in C as part of its assessment structure. Students must work individually on four projects over the course of the semester, which together comprise 45% of their final course grade. Grading projects involves running students’ compiled code against automated tests that determine whether a student program’s output matches the instructor’s canonical output. If a student’s program completely matches the canonical output, the student receives at least a 90% grade on a project. The remaining 10% are discretionarily allocated “style” points, awarded for things like proper formatting, code commenting, and functional decomposition (Field Notes).

This study centers on “Flights Database,” the second of four projects assigned to Intermediate Programming students during the spring 2012 semester. Students were asked to build a text menu-based program that would let users query information about airports and plan non-stop and one-stop flights between airports. For this project, the instructor gave students three separate text files as source material. The **airports** file contained names of airports and their three-letter abbreviation codes; the **routes** file contained 3-tuples of two airport codes and the route number of a flight flying between them; the **flights** file contained a list of specific flight information (including arrival and departure times) by route number. Crucially, in order to be able to respond to user queries students would need to build a program that could coordinate information across all three files to return an answer.

My analysis details the work of Rebecca, a female first-year electrical engineering major. I focus specifically on Rebecca’s code for finding “one-stop” flights, which the instructor defined as “all pairs of ﬂights that route the user between the departure and arrival airports with exactly 1 stop (i.e., a one-connection ﬂight)” (Flights Database handout, 2012). The one-stop problem is particularly challenging. To solve it successfully students’ code must accept a user’s choice of airports and day, then stitch together routes that involve two separate flights in a way that passes stringent constraints for acceptable layover times.

## Research question and methods

This study proceeds from an empirical challenge: how can we develop accounts of students’ programming activity that explain the form and evolution of their code on a design project? The focused form of that challenge for this study is “how can we understand the unconventional design choices embedded in Rebecca’s one-stop flight code?” To answer that question, my study draws from three data streams: ethnographic observation, clinical interviewing, and code snapshot analysis.

For two semesters, I ethnographically embedded myself in the same instructor’s section of Intermediate Programming. My aim throughout was to see what students see in terms of course material, assignment directives, and instruction. In fall 2011 I observed approximately 50% of the course lectures. I also independently completed all class homeworks and three out of the four course projects to more fully understand the course’s assessments. In spring 2012 I continued attending lectures, though less frequently, and began attending select TA-led discussion sections. During both lectures and discussion sections I took field notes while recording ambient audio using a LiveScribe Pulse pen.

Rebecca was one of four students (three female, one male) I solicited to participate in a series of 1-hour outside-of-class clinical interviews during the spring 2012 semester. I interviewed Rebecca five times, and in typical interviews I split time between asking about her experiences in the course and giving her time in the interview to work on her project code. During each interview, I simultaneously used:

* A Kodak Zi8 camera for video-recording our interactions
* A LiveScribe Pulse pen to capture Rebecca’s on-paper penstrokes
* A MacBook Pro (early 2011) with screen-recording software to capture everything on-screen while Rebecca programmed

The final component of data gathering is modeled after Jadud’s (2006) system for capturing students’ code. My colleagues and I developed software, built around the open-source version control system called Git, that effectively creates an entire copy of a student’s code—what we call “snapshots”—every time students compile their code. Our software then sends those snapshots to a secure, researcher-accessible server in real-time as they’re created. Consequently, I could plan each interview with Rebecca around up-to-the-minute knowledge of her work—in some cases work she had completed just hours before the interview—and tailor my interview questions to emerging patterns in her code. In total, Rebecca’s work resulted in 958 compilation snapshots over the course of the semester.

In what follows, I snapshot and interview data to explore the form and history of certain features of Rebecca’s one-stop flight code. First, I simulate an analysis of Rebecca’s code in the methodological vein of Jadud (2006) and others (Rodrigo, Tabanao, et al., 2009; Tabanao et al., 2011), analyzing only Rebecca’s code snapshots. In this first analysis, I focus on the form and evolution of particular design features of Rebecca’s code. Then, I present a second, complementary analysis using contextual data from my clinical interviews with Rebecca. In my second analysis I go beyond the snapshots to highlight why, in Rebecca’s own words, she made those particular design choices.

# Code Snapshot Data and Analysis

In this section, I simulate what data on Rebecca would look like using an existing method of analysis described in {Jadud, 2007}. Namely, I work at a level of remove from Rebecca herself. I base my inferences only on her actual code, limiting my interpretations about what she may have *intended* to do or *designed* her code to do only to what’s supported in the snapshot-by-snapshot record of her code. Through this “snapshot-only” analysis, I show that we can see in great detail how Rebecca’s code embodies particular design features that diverge from how an expert might have handled this design task. I restrict my analysis here to just two such features of Rebecca’s code:

1. Rebecca’s use of repeated file-scanning loops to create a depth-first search algorithm
2. Rebecca’s 7-fold duplication of code to handle checking for flights on each day of the week

## Rebecca’s code uses nested scan loops

After declaring variables and opening the three provided text files (flights, routes, and airports), Rebecca’s one-stop flight code enters a series of conditionally-nested `fscanf()` commands. The purpose of `fscanf()` is to scan through the characters of a file, most often one line at a time. The programmer can specify patterns of text to look for, e.g., “in each line, look for a word, followed by a space, followed by two digits.” `fscanf()` also gives the programmer the flexibility to store such matched patterns. “Once you find a word followed by two digits, store the word in the following location in memory.” What's interesting isn't *that* Rebecca used `fscanf()` to read through files like the list of airports. Any suitable solution would need to read through those files. Rather, what's interesting is *how* she uses `fscanf()` in her design.

Rebecca’s file-scanning logic never persistently stores the contents of the files it reads in. Instead, her program reads through files one line at a time, and it essentially can't cognize about airport/flight information not in the line currently being scanned. All it can do is copy single patterns temporarily, then rewind the file back to the top and start reading in one-line-at-a-time again.

What’s consequential about Rebecca’s design choice? From a design standpoint, her code has to repeatedly open multiple files and scan them one line at a time in order to coordinate information. So, the task of finding a one-stop flight between two cities becomes a series of repeated, one-line-at-a-time scans of external files:

1. Scan the **airports** file to check the correctness of the user’s input (line 20)
2. Scan the file of pairwise airport **routes** (line 24)
3. To find possible connection cities, scan the **routes** again (line 38)
4. If a route matches, scan the **flights** file to verify whether the time/day constraints are acceptable (one of the following lines depending on the chosen day: 52, 79, 106, 134, 162, 190, 218).

Rebecca’s code is both visually and computationally complex. The multiply-nested blocks can make it difficult for a human reader to follow the code flow, which may have made it challenging for Rebecca to debug her own work. Moreover, nested for- and while-loops increase the complexity space of a program—what computer scientists call the big-O characterization of her algorithm. For every nesting of a scan loop (there are 4 here) Rebecca increases by 1 the degree of a polynomial that represents the execution time of her program. So, from a performance perspective, Rebecca’s design suffers a trade-off in that with each invocation of a scanning loop, we see a geometric increase in the time it takes her program to execute.

A second feature or Rebecca’s `fscanf()` design is that it uses abstraction in a non-obvious way. One design solution to the problem—and one seemingly dictated by the assignment—is to create three separate arrays in the computer’s memory:

To parse the 3 airline ﬂight database ﬁles, you will need to declare arrays that will receive all the data. For the purposes of determining array sizes, you may assume there will never be more than 100 airports in the “airports.txt” ﬁle, 500 route IDs in the “routes.txt” ﬁle, and 3000 ﬂights in the “ﬂights.txt” ﬁle. (Flights Database class assignment, 2012)

Presumably, from the instructor’s directive, one “will need” to have an array of airports (mapping 3-letter code to full airport name), an array of routes (mapping a pair of airports to a unique routing number), and an array of flights (mapping unique flight numbers to a collection of information about that flight). Rebecca creates no arrays. Instead, her code forces her scan loops to accomplish the same task that a memory-persistent data structure would, only without the persistence. A consequence of Rebecca’s approach is that she has no way to refer to arbitrary airports, routes, or flights in memory, since her program has no mechanism to store such information persistently. A second consequence is that since she avoids persistent data structures, the complex work her program does to read through each line of each file (above) has to be repeated every single time a user initiates a query.

## Rebecca repeats the same chunk of code seven times

The second feature that strikes one as unusual about Rebecca's code is how she handles processing different input cases for different days of the week. Because users choose a day to fly (and some flights only run on certain days), students’ code must be able to handle each of the seven possible cases for when a user would want to fly. In principle, Rebecca's code achieves just that.[[2]](#footnote-2) She's written conditional code that could execute uniquely to handle each of the seven possible inputs. What stands out is how much word-for-word duplication occurs in Rebecca's code.

A second key feature of Rebecca’s code is the almost identical repetition of a single 23-line code chunk seven times (lines 50–240). {The figure below} represents a side-by-side delta-comparison of two code blocks for checking day-based input. Lines 158–184 of Rebecca’s original code are on the left; lines 214–241 are on the right, and in the figure lines have been renumbered to ease comparison. The two code blocks demonstrate just how much code is duplicated for checking user input based on days of the week. Between these two chunks there are only three differences: all of the references to day are changed from 5 (on the left) to 7 (on the right). Moreover, the changes from block to block are patternistic and predictable: the third line of each block just checks whether the rest of the block should run, while the eleventh line of each block uses a numerical value that's always the same as the day being checked. Everything else becomes duplicate boilerplate. This procedure is essentially repeated 7 times; once for each day of the week.[[3]](#footnote-3)

But how did this particular repetition of code come to be? An advantage of our research methods is that we can pose and answer such questions, at least instrumentally. We use git to trace the provenance of individual lines of code and discover where, in the history of a student’s work, those lines of code were first created and later changed. In the case of this section of Rebecca’s code, two snapshots are helpful. {The figure below} shows those two snapshots, which occurred on March 22, 2012, approximately one hour and twenty minutes apart. Identifying information has been redacted.

{Snapshot picture}

The key lines in the left snapshot are lines 18–25, which form the skeleton of code that checks each of the 7 possible days the user could input. In the snapshot on the right, we see Rebecca had begun fleshing out the code for checking each day. Crucially, she also left in placeholder text: the `if (\* day has` code, which appears on lines 25, 44, 59, 74, and so on, for each day of the week.

Over the next six days, the code for checking days continued evolving beyond its original skeleton. From the history, we can see that Rebecca tried implementing code for a separate `check\_days` function to supplant her placeholder `if (\* day has` code. Later, however, calls to that function were commented out (effectively making them inactive). The final assignment she submitted includes the code for her check\_days() function, but calls to `check\_days()` remain commented out, so the function itself is never used. Moreover, a pattern that stands out is that in many snapshots, we see Rebecca making the same *change* to the code seven times. Such evidence suggests—but does not prove—that Rebecca may have written a change, then copied and pasted that change six more times for each subsequent day-specific code block.

In closing, a strong claim we can make from Rebecca’s code is that it doesn’t make extensive use of C’s features for creating abstractions. In one instance, it avoids a direct requirement in the assignment (to create memory-persisten arrays for holding data), and instead uses intricately nested scanning loops. In another case, it avoids the principle of functional abstraction to express repeated procedures, as Rebecca’s sevenfold-duplicated weekday computations show. In short, looking only at the code and its history, it appears she has created a lengthy, complex design that avoids the use of core concepts covered in class.

It's worthwhile to ask *why* Rebecca's code reflects a design that both ignores assignment directives (using scan loops instead of declaring and using an array) and violates what's putatively a core tenet of the class: using abstraction as a way to manage intellectual complexity. In the next section, I draw from 5 clinical interviews with Rebecca (totalling over 5 hours) to make the case that particular factors--personal, historical, and emotional--drove the design decisions that resulted in her unusual file-scanning logic and repeated-code computations.

# Interview Data and Analysis

In the previous section, I described two unusual features of Rebecca’s code for searching one-stop flights:

1. Her use of multiply nested loops that scan through source information files *without* storing the information in those files persistently in long-term memory
2. The code for handling a user’s chosen day, which was essentially the same block of code copied and pasted 7 times

In this section, I select data from over five hours of clinical interviews I conducted with Rebecca to detail explanations for Rebecca’s design choices. For features 1 and 2, I draw from data to explain how design decisions that might seem unusual to an expert in fact grew rather unproblematically (for Rebecca) as ways of deliberately transferring prior knowledge and designs (1) or coping with a constraint to produce a reliable solution she could trust (2). I conclude this section with the questions that motivate us to move outward from the data to its implications for methodological and theoretical perspectives in computer science education..

## Rebecca used fscanf loops because she framed the problem as one of code reuse from a similar task.

My first opportunity to discuss Rebecca’s one-stop flight work was on March 16, 2012, in what would be her third of five interviews that semester. We were discussing her prospective design plans for the Flight Database project. As Rebecca began explaining how the logic for a one-stop flight search was supposed to work, she described what she saw as one of the central difficulties of the project: that relevant information was spread across multiple files (Interview, March 16, 2012).

As Rebecca explained, something as simple as finding a flight from, say, JFK to BWI “involves scanning through multiple files, because it’s not like one file that has everything conveniently like there” (Interview, March 16, 2012). When I asked what would make things easier if, hypothetically, all the information she needed *were* in one file, Rebecca appealed to a previous assignment from last semester. In Rebecca’s first-semester programming course (Beginning Programming for Engineers), one of several multi-week projects had students creating a system for users to conduct a fantasy football draft. The project involved, among other things, topics related to basic file management in the C language, including how to read in a file from disk (Interview, March 16, 2012). Rebecca explained:

Rebecca: Um, like, cuz when we first got this project, uh, I actually was thinking “oh, well this is just a lot like our fantasy football project we did last year.” /Huh/ We uh, had to scan in, uh, someone had to enter like “I wanna pick a quarterback,” so then you had to scan in and go and look for all the quarterbacks in the file and say “OK, this is the quarterback” and everything. But, in that project we only had the one file that had everyone listed: quarterbacks, runningbacks, wide receiver, in one file. And all the information you needed there /Mmhmm/ So, you could just get it all and compare it all at once with one scanf /mmm/ whereas this you have to, take, uh, you scan in the flights, ah, the flights file. So, then you find the flight number. You have to save the ID from that flight number, use that ID to scan into the routes file /mmhmm/ and then save the routes information and then print it out with the flights information. (Interview, March 16, 2012)

Rebecca’s comments suggest she saw a coupling between the arrangement of the input information and the structure (and complexity) of the computational logic needed to process it. When one fantasy football file contained all of the relevant information (player, position, team, etc.) it could be read in and processed one line at a time. When information was fractured across files (pairs of airport codes in one file, full spell-outs of airport names in another, for example), Rebecca realized that she’d need to use information shared across files (such as a route ID) to coordinate a scan across one file with a scan across another (and possibly an additional scan across a third file) before she would have all the necessary information and computations to return a result.

I was interested in the connections Rebecca saw between her previous fantasy football project and the current flights project, so I pressed on

Interviewer: So, did you ever think about, like, I mean based on that, like, man, fantasy football project--so sure, it was in 140 definitely, um, but, uh, it was easier because all the information was in one place. /mmhmm/ Um, did you ever think about like, is there a way I could make this project, like /Oh!/ like the fantasy /Oh yeah, definitely!/ football project. (Interview, March 16, 2012)

Rebecca’s elaborated answer was emphatic.

Rebecca: That was like, as soon as we got this project I was like, ah! fantasy football! I'm just gonna go and see how much code I can rework from that and like, use /mmhmm/ in this project. And, my whole main file, like all those NULL checks and everything, I mean they're really simple to write, but I just copied 'em and put 'em there, cuz, we had the same thing. /Mmhmm/ Um, just changed, like, the names of the files.

As she explained, “reading in files” was a topic covered extensively in Introductory Programming—the first course of the sequence—but they hadn’t talked much about it in this semester’s course.

Rebecca: Going back and I was looking at how I scanned in the information from the files, cuz, we haven't really done anything like that this semester. Uh, scanning in from files before--

Interviewer: Mmhmm. Oh. /But/ You--you hadn't done that, um--had you done it before at all? Or just not this semester?

Rebecca: Yeah. Just not this semester /Oh, OK/ We did it alot last semester, like the whole fantasy football project and pretty much every project after that. But, uh, just this semester because it was a big {Basic Programming} topic we hadn't talked about it much. /Yup/ So, I just, uh, went back to check how I did that /mmhmm/ and then, if I could I copied, but because a lot of the variables were different, uh, like these were more var--less, less variables, and more strings than last year /mmhmm/ uh, I just retyped it out. I just looked at how it was similar. (Interview, March 16, 2012)

In sum, then, Rebecca’s repeated, nested scan loops were actually lifted from a previous semester’s project. By her telling, what seemed obvious was that “scanning in from files” was a topic they’d already covered, which meant she’d already developed a workable solution for how to solve that problem. Thus, she saw the new problem of how to coordinate airport information from different files as a new instance of the old problem of reading information in from one file. Her work, accordingly, became about opportunistically adapting a previously working solution to fit the current circumstances.

## Rebecca repeated code because she wanted to re-use functionality she could trust

By our interview on April 6, Rebecca had already completed and submitted her code for the flights database project. When I looked at the final form of her code for finding one-stop flights I noted an unusual pattern described in {section XX} above: she had a code chunk repeated almost character-for-character 7 times. In the interview, this section is what Rebecca referred to as “my obnoxiously long part of my code” (Interview, April 6, 2012):

Rebecca: So, the way I did it was really long and probably, there was probably like a much easier way, but I just did a giant if--if statements {swings cursor from line 50 to line 63} If they wanted to fly on Monday /OK/ I went through and checked to see if the route ID was the same {wiggles cursor across line 54}, and if it did, I went through to che--uh, I made a check\_days function {wiggles cursor across line 60} uh, I ended up commenting that out cuz I didn't end up /mmhmm/ finishing it. But, uh, my check\_days function worked, it just didn't work completely with the code /OK/ (Interview, April 6, 2012)

As I scrolled the screen to look at each of the repeated blocks of code, Rebecca elaborated:

Rebecca: And this is why my code, I feel like, is not uh, concise enough, or, I don't really, I forget the word they use, but uh /{inaudible}/ it's very long because I couldn't figure out if I should do a while loop or whatever /Uh-huh/ But, so I was just like, I know this way should work if I get everything else right, that uh, just go through, if input's 1, if input's 2 and just do the same thing in each of 'em just /Mmmhmm/ check for, “oh, if days is 2, if days is 1” instead of, like--Cuz I probably could have done, like, maybe a giant while loop, um, to try and, and if, while, inputs something, uh, then you check to see whatever i is. But, I could, I didn't--couldn't figure out how that would work, so I just did the same thing six times.

Interviewer: So, in, in each one of these it's like, looks, and I'm not sure about this, but it looks like the way you wrote it--so this {highlights line 79} is pretty much the same in all of them, right? /Yes/

Interviewer: So's this one {highlights line 81} /Yes/ this one {highlights line 83} Here's where it's different {highlights line 85}

Rebecca: Yes, because it just checks if it's a 2 instead of a 1.

Interviewer: OK. Um. /And then everything else is still the same/ Layover's still the same. OK.

Rebecca: Yeah. So that's why it's prob--it's not, uh, the neatest code or whatever, because it's the same thing six times. (Interview, April 6, 2012)

Given Rebecca’s assertions that her code wasn’t neat, I asked what, if anything she might change if she hypothetically had another week to work on the project.

Rebecca: Um, first I'd try and get it to make sure it worked completely /Ahh, OK, yeah/ this way, {laughs}, uh, and then, if I had the week after that whatever, I'd probably go through and see if I could figure out a way to make it concise-r because he likes uh, neat, as, like, code that's, uh, easy for the user to see /uh-huh/ I guess. Uh, I forget what, I keep forgetting what the word he used was at the beginning of the year, but uh, just very concise and, uh, this is [a] very expanded {laughs} way of coding, but, it made sense to me at the time and I was just like “I just want something that makes sense right now.” /Right/ So, that I can actually work with and have an idea.

Interviewer: Um, OK. So, so it would take you some extra thinking to figure out /Mmmhmm/ how to break this down into /Yes/ smaller stuff /smaller code/ Do you feel like you've had a lot of practice doing that, or like?

Rebecca: Uh, a little. Like, but, a lot of times in [Beginning Programming] they didn't really mention too much about being concise. They were just like "if you can do it, do it" {laughs} /OK/ So I usually stuck to what made sense to me /Right/ uh, to turn the projects in. (Interview, April 6, 2012)

In summary, Rebecca’s “expanded way of coding” was a way of expressing ideas in code that, in her own words, “made sense to me.” Moreover, her Beginning Programming course seemed, to her, to set expectations that functionality comes first; “neatness” second. If she hypothetically had more time to work on the project, her first priority would be to get her existing code working. Consequently, Rebecca’s repetition of code can be understood as a kind of pragmatic solution to a difficult problem: choosing which computational techniques were best for accomplishing a complex goal. Ultimately, she chose repeating code that made sense to her over the difficult-to-envision alternative of a “giant while loop.”

# Discussion

## Does Rebecca “Know” procedural abstraction

## Why is inferring from snapshots difficult?

## What kinds of questions are we trying to answer?

Fincher, S., Tenenberg, J., & Robins, A. (2011). Research design: necessary bricolage. *Proceedings of the seventh international workshop on Computing education research*, ICER ’11 (pp. 27–32). New York, NY, USA: ACM. doi:10.1145/2016911.2016919

Jadud, M. C. (2006). Methods and tools for exploring novice compilation behaviour. *Proceedings of the 2006 international workshop on Computing education research - ICER ’06* (p. 73). Presented at the the 2006 international workshop, Canterbury, United Kingdom. doi:10.1145/1151588.1151600

Papert, S. (1980). *Mindstorms: Children, Computers, and Powerful Ideas*. New York: Basic Books. Retrieved from http://lccn.loc.gov/79005200

Rodrigo, M. M. T., & Baker, R. S. J. d. (2009). Coarse-grained detection of student frustration in an introductory programming course. *Proceedings of the fifth international workshop on Computing education research workshop*, ICER ’09 (pp. 75–80). New York, NY, USA: ACM. doi:10.1145/1584322.1584332

Rodrigo, M. M. T., Baker, R. S., Jadud, M. C., Amarra, A. C. M., Dy, T., Espejo-Lahoz, M. B. V., Lim, S. A. L., et al. (2009). Affective and behavioral predictors of novice programmer achievement. *Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education*, ITiCSE ’09 (pp. 156–160). New York, NY, USA: ACM. doi:10.1145/1562877.1562929

Rodrigo, M. M. T., Tabanao, E., Lahoz, M. B. ., & Jadud, M. C. (2009). Analyzing Online Protocols to Characterize Novice Java Programmers. *Philippine Journal of Science*, *138*(2), 177–190.

Spacco, J., Pugh, W., Ayewah, N., & Hovemeyer, D. (2006). The Marmoset project: an automated snapshot, submission, and testing system. *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA ’06 (pp. 669–670). New York, NY, USA: ACM. doi:10.1145/1176617.1176665

Tabanao, E. S., Rodrigo, M. M. T., & Jadud, M. C. (2011). Predicting at-risk novice Java programmers through the analysis of online protocols. *Proceedings of the seventh international workshop on Computing education research*, ICER ’11 (pp. 85–92). New York, NY, USA: ACM. doi:10.1145/2016911.2016930

1. Hereafter, I use “Intermediate Programming” to refer to the second course and “Beginning Programming” to refer to the first. [↑](#footnote-ref-1)
2. I say “in principle” because Rebecca’s code would not compile on my machine. So, in practice, her design contains compile-time errors (and possibly run-time errors). Nevertheless, her code provides ample evidence that she was attempting conditional logic to handle each possible day. [↑](#footnote-ref-2)
3. I say “essentially repeated” because, in fact, there are minute differences in each of the code chunk’s incarnations. Line 218, for instance, has `d\_letter` as a variable in the scanning pattern; line 190 omits `d\_letter.` [↑](#footnote-ref-3)