Mining compilation data to better prepare and assign K-12 coding mentors

Chinmay Sheth, Vaitheeka Nallasamy, Kruthiga Karunakaran, Stephanie Li, Yiding Li and Christopher Kumar Anand

McMaster University, 1280 Main St W, Hamilton, ON L8S 4L8, Canada

Abstract

Our university outreach program has introduced over 30,000 Grade 4 to 8 students to functional programming in Elm over the last decade. Pre-pandemic, mentors would visit students in-person classrooms to conduct workshops aligned with their curriculum. With the advent of the pandemic, this switched to virtual visits, which allowed us to teach children in remote regions and other countries, greatly increasing our reach. Further increasing our reach will require smarter use of most important resource: coding mentors. In this retrospective study, we looked at statistics for compilations on our web-based integrated development environment in order to identify patterns that could immediately inform mentor training programs, and in the medium term be used to optimize resources by directing mentors to the students most in need of help, even when they have not explicitly requested help via the built-in mentor chat.

Keywords

computer science education, introduction to programming, teacher dashboard

1. Introduction

Learning to Code has been called the Literacy of the 21st Century. Bers [1] asks "What is literacy? It is the ability to use a symbol system (a programming language or a natural written language) and a technological tool [..] to comprehend, generate, communicate, and express ideas[...]" Among the reasons we care about it: "Literacy ensures participation in decision-making processes and civic institutions. Those who can't read and write are left out of power structures. Their civic voices are not heard." At a time when someone will be making decisions about how software is used and how it therefore impacts society, it is important that the majority be capable of participating in the debate. Burke et al. [2] amplify this sentiment: "If a learner reaches adulthood and cannot read or write, it is generally identified as a collective societal failure. As society is increasingly digitized, students need to be able to read and understand the information contained in code".

Whereas Bers [1] argues for the importance of coding tools for children as young as kindergarten, for whom they point to the success of block coding and tangibles; addressing the need to "read and understand the information contained in code" at an adult level probably requires understanding of textual code. It is therefore important to make text-based coding easier to learn. In our outreach programming, which has visited thousands of classrooms and introduced over 30,000 children to coding, the bottleneck to growth has long been our ability to train mentors to visit classes in-person or virtually. Anecdotally, we know that primary teachers are very reluctant to introduce text-based coding about which they have insufficient training, but enough word-of-mouth evidence that even children doing the simplest things can cause compilers to emit cryptic error messages.

How can we better train undergraduate mentors now, and, in the future, primary teachers? This paper looks at the data generated by our Web-based Integrated Development Environment (WebIDE), to see if it offers useful insights in training future instructors. Our WebIDE is backed up by a server

STEM@ICon-MaSTEd 2025: 4th Yurii Ramskyi STE(A)M Workshop co-located with XVII International Conference on Mathematics, Science and Technology Education, May 14, 2025, Ternopil, Ukraine

shethc@mcmaster.ca (C. Sheth); nallasav@mcmaster.ca (V. Nallasamy); kk@mcmaster.ca (K. Karunakaran); li3424@mcmaster.ca (S. Li); liy443@mcmaster.ca (Y. Li); anandc@mcmaster.ca (C. K. Anand)
https://www.cas.mcmaster.ca/~anand (C. K. Anand)
0009-0007-7723-7611 (C. Sheth); 0009-0005-5359-9277 (S. Li); 0000-0002-7863-8595 (C. K. Anand)

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

running the Elm compiler. To allow children to return to previous versions of their code and support our help system, all submissions are stored on the server every time the compile button is clicked. The help system consists of a chat pane where children can ask for help. Mentors can edit and compile a copy of the children's code to understand the problem and double-check the proposed solution. This is especially helpful for in-class workshops where screen sharing is not possible. But children do not always ask for help when they need it. Analyzing data on compilation results could be used in two ways: (1) to better train mentors for the types of problems children will encounter, (2) provide mentors with real-time feedback on which students need assistance, and (3) improve resource allocation as an organization. As a first step toward these goals, we asked two research questions:

RQ1 Are compiler error types correlated with activity type?
RQ2 Does time to resolve compiler errors follow a statistical distribution?

Another feature of our original WebIDE led to a natural experiment. Namely, students were presented with a "slot" system for storing their code, referencing the slot systems many game systems use for storing games. The slots were grouped into 10 per activity type, and the basic activity types mirrored the initial lessons in our outreach program, so they are a good proxy for student experience.

1.1. Contributions

This paper contributes to STEM education in three ways:

1. Understanding syntax and type errors of beginning programmers will lead to more effective introductory Computer Science teaching, which is one important Science.
2. The Algebraic Thinking curriculum being used in these code examples was developed to help children solidify their arithmetic and geometric knowledge and prepare them for high school algebra, all of which are core Mathematics subjects.
3. Finally, Silver et al. [3] have identified success in high school algebra as the bottleneck to success in high school and a gateway to STEM education pathways, so making progress here makes all higher STEM subjects more accessible.

The specific contributions of this paper are the identification of the power law as best fitting the distribution of time to fix compilation errors, and the observation that error types rapidly change over time as young students learn to code in Elm. The remaining sections of this paper discuss related work, the background of our outreach program, methods, results and conclusion.

2. Related work

There is a long history of applying analytics in education [4]. Of most interest to computer scientists are the linked literatures on Open Learner Models and Analytics Dashboards. Open Learner Models (OLMs) arose out of the attempt to capture some of the performance advantage of tutoring over classroom teaching by using software tutors [5]. The simplest OLMs represent progress through a pre-defined curriculum with accuracy on quiz questions and problems. In some cases, OLMs plot individual progress against class averages. Learning Analytics Dashboards (LADs) "make use of data science methods to analyze data and report the results" [6]. There is a lot of overlap between the two. Since LADs are supposed to improve learning, there is a growing movement to incorporate educational theories into their design [6].

Starting with a different problem, our approach is following a different trajectory. Our approach could be called "mentor in the loop", in that we are not trying to synthesize a software tutor, but to increase the effectiveness of the mentors we already employ. Synthesizing software tutors is difficult, at least for introductory high-school algebra, but has been shown to be effective [7]. It is rewarding and difficult research. On the other hand, using near-peer mentoring in computing education has a secondary benefit that "youth's interest in computer science (CS) can be sparked by providing them with role models who are relatable and who resonate with their identities" [8]. This may be true in other areas, but we know that in software, the rapid growth of the field guarantees that the number of role models in the average community or family network will be small compared to the learning and career opportunities. This will be exacerbated in economically disadvantaged communities. So it makes sense to design systems to support near-peer mentors. Unlike normal teachers, near-peer mentors cannot be expected to have any knowledge of educational theories (other than the discredited "learning styles"). Our trained mentors spend from 10 to 100 hours mentoring, and a high turnover is acceptable, because it is also a learning experience for the mentors. We are very confident that mentoring helps them improve their communication skills, and we suspect that it also improves self-efficacy and metacognitive skills in the mentors, but we have not studied this, and it is not needed to justify our program. While we would like to develop an educational theory or interpret an existing theory in our context, and explain the student characteristics blind clustering appears to identify, that is a long-term goal. In the near future, we can measure the effectiveness of our LAD in terms of student output.

2.1. Functional programming

Since our outreach program adopted Elm, a functional language for teaching, it is important to mention some facts about Functional Programming. Functional programming [9] is a value-oriented programming paradigm, consisting of functions. Functions consume and produce values. There are no loops, and conditional expressions replace conditional statements, but functions are first-class values and can, e.g., be passed as parameters. There are two variations in functional programming languages: (1) typed or not and (2) eager or lazy. These variations lead to differences in programming style.

Many non-functional programming languages are adopting functional features, including Scala, Swift and Python.

Krishnamurthi and Fisler [9] agree with the common perception that writing programs in imperative programming languages is much easier as the state provides convenient communication channels between parts of a program, but this makes reasoning and debugging harder, whereas on the other hand functional programming has the opposite affordances. Students studying object-oriented programming are taught different skills and programming styles which reveal that the way of approaching programming and problem-solving differs in students studying different paradigms. Functional programming students perform better by having high level structures and and composing solutions out of simpler functions than object-oriented students who try solving the entire problem in a single traversal of data. They also use built-in/higher order functions to implement subtasks which performed multiple passes over input data and had to release unwanted memory for intermediate data. Functional programming students create short functions for specific tasks, which create intermediate data. They also use `filter` and `map` rather than loops and non-general library functions. Thus, we should expect that a student who learns Java after learning functional programming may well program with different patterns than a student whose prior experience was entirely imperative.

Note that our experience is that functional programming with appropriate supports is easier for Grade 4 to 8 students. Prior to using Elm, we had developed coding activities using Python, and there was a dramatic improvement in student focus after we made the change.

2.2. Elm language

Elm (https://elm-lang.org) is a functional language designed for the development of front-end web applications [10], and sold to front-end developers as a way of avoiding the many software quality issues which plague JavaScript programs. Its syntax, based on Haskell, is intentionally simple. For example, it has no support for user-defined type classes. In addition to strictly enforcing types, the Elm compiler also forces programmers to follow best practices, such as disallowing incomplete case coverage in case expressions. Elm apps use a model-view-update pattern in which users write pure functions and the run-time system handles side effects without the need for advanced concepts. Elm code compiles to JavaScript, simplifying deployment and visualization.

While many consider that functional programming should be reserved for expert users, many of the features useful for experts (strict types, pure functions) are also very useful for beginners. In addition to the practical implications of compiling to JavaScript, Elm's combination of simple syntax, strict typing, and purity which matches students' pre-existing intuition about math proves to be an asset to our program. These features allow the development of tools and curricula which would not otherwise be easy or possible in an imperative language with side effects such as Python.

In another conference presentation, we present parallel work on supporting non-English-speaking learners in a tool called ShapeCreator. The use of function composition to structure code which is favoured by functional programmers is evident in the graphical layout of ShapeCreator, and it matches the way geometry is taught in the early grades.

3. Outreach program

3.1. McMaster Start Coding Program

Our Outreach Program has been operating for the past decade. A mainly volunteer group of undergraduate and graduate students develop lesson plans and deliver free workshops to schools, public libraries, and community centres in the Hamilton, Ontario, Canada area [11]. During the COVID-19 pandemic, the program shifted online and has taught an increasing numbers of students. The goal of the program is to foster interest and ability in STEM subjects through coding, especially for those groups who are underrepresented in STEM subjects, such as girls and underprivileged youth.

To support these workshops, we have developed several tools, including:

1. An open-source Elm graphics library [10], GraphicSVG[1].
2. An online mentorship and Elm compilation system incorporating massive collaborative programming tasks, including the Wordathon[2] and comic book storytelling[3].
3. A curriculum for introducing graphics programming designed to prepare children for algebra [11].

3.2. WebIDE

In our IDE, we deliberately do not store any personally identifiable information, so this dataset includes children who programmed during one class visit, a series of class visits, one or more summer camps, training sessions for potential undergraduate mentors and first-year undergraduate students.

Using a video-game metaphor, children have access to different coding environments, originally with 10 slots available for different modules. Once they pick a slot, they see an interface with four quadrants, for their code, graphical output or compiler errors, help chat, and (optionally) additional information about the activity. If additional information is not available for a slot, half the screen is devoted to their code. Different slots hide some or all components in a working program. For example in *Picture* slots, the main function is hidden and no interaction is possible, instead children must define a `myShapes` top-level definition whose type must be a list of shapes. In *Animation* slots, `myShapes` is a function with an input which is a record with a single field, the current time in seconds since the program was "played". Depending on the level of the class, they may only learn to use the Picture slot, or they may advance to the Animation slot in their second session. A *Wordathon* is a special activty in which children are assigned beginner reading words to code as pictures or animations, and their code is combined together into a reading game. Sometimes classes compete against each other to win a pizza party. Teachers like these activities because their students both feel like they are doing something useful, and get to compete in a fun competition—and they love pizza! The *Wordathon* slot as a second definition, `myWord` of type `String`, where they must specify the word they are creating, and which will be used when their module is imported into the game module. It also has a semi-transparent background masking out the border of the output which will be masked out in the game. This is especially useful if

[1] https://package.elm-lang.org/packages/MacCASOutreach/graphicsvg/latest/GraphicSVG

[2] http://outreach.mcmaster.ca/#wordathon2019

[3] http://outreach.mcmaster.ca/#comics2019

they want to animate an object sliding into view. Although there are many advanced slot types, *Game* is the fourth commonly used slot type. It adds interactivity, which requires the definition of a message (event) type, as well as a model type (which in previous slots was hard-coded to be a record containing only the animation time). Only classes doing extended workshops used the *Game* slots. This allows us to infer a lot about the experience and immediate goals of the users from the slot type they are using.

4. Method

In order to be able to give mentors access to modules with pending help requests, all versions of all modules are left on the file system of the server. Each module is stored using the childrens' randomly-assigned IDs, timestamp, and slot number (which encodes the activity type in the most significant digits). We extracted 254,708 compilation attempts from 5330 users, from 2019 until January 2022. Each of the code fragments was recompiled together with the hidden boilerplate code to extract the success/error and error code results, and the results were added to a database. Python scripts were used to calculate time between first error of a user in a slot and the next successful compilation, and add this information to the database. This data was then compared to known probability distributions and compared visually.

Because errors include excerpts from code, this database was further reduced to the time, slot, user id, and error type (not the whole error message) for errors. It was further cleaned to remove all user ids corresponding to teachers, mentors and undergraduate students. This cleaned data contains 13012 rows, and is available for download as an Excel file together with the pivot tables used in the analysis.

Next, the dataset with error types and timestamps was used to simulate a mentor dashboard in which errors appear as they occur.

Finally, we investigated the possibility of providing additional feedback by comparing compilation behaviour with past observations. To this end we collected compilation results by user, resulting in a high-dimensional dataset, to which we applied t-distributed stochastic neighbor embedding (t-SNE) for dimensional reduction, followed by k-means clustering. This produced very interesting-looking complex clusters, but there were too many clusters for us to develop meaningful interpretations of the results, so we do not include them in this paper, but hope other researchers will have greater success in extracting meaning from this approach.

5. Results

We were able to derive concrete answers to our research questions, and develop a mentor dashboard which could provide mentors an at-a-glance view of the time to resolve errors.

5.1. Predicting the time to resolve an error

To try to answer RQ2, our first attempt to predict when students might need help was to model the time between an error occurring and the error getting resolved. Figure 1 shows that the time to resolve a compiler error approximates a power law distribution [12]. Even though most of the errors in the distribution took less than a minute to resolve, there are many errors which took considerable amount of time to resolve. Knowing this distribution, we can predict the time it would take a student to resolve an error and the chance they will resolve it on their own in a given amount of time. This distribution was determined from pre-2019 compilation data, but we believe the pattern will hold in the present dataset. If we assume that a power law is always a good approximation for time to correct an error, we can re-estimate model parameters for subsets of users or subsets of sessions.

5.2. Error types

Figure 2 shows the distribution of the top 10 errors encountered by children in the four basic slots. Most of the data is associated with the basic slot types, (Picture, Animation, Game, and Wordathon slots),

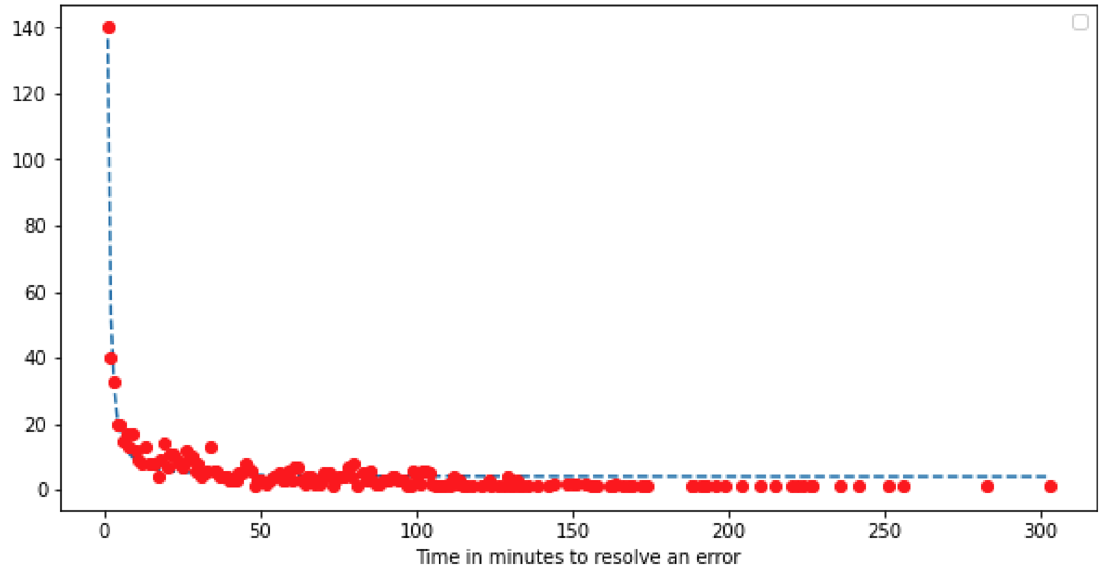


Figure 1: Fitting the data to a power law distribution.

with a significant number of compilations of advanced and/or experimental activities, which have been excluded due to the difficulty of separating different use cases.

To try to answer RQ1, we explored error statistics using different visualizations, and discovered many patterns. Figure 3 shows that Unfinished List is the most prevalent error, but this is heavily weighted to *Picture* slots. This can be explained by the fact that in early lessons, students must be taught the syntax for lists, which fortunately follows English grammar in requiring commas between elements, but unfortunately also requires matching square brackets around the list which students have not encountered outside of a text-based language. Fortunately, we see that the number of such errors declines dramatically as students advance from *Picture* to more advanced slots, and this is in spite of the counfounding factor that *Picture* slots are sometimes used in more advanced projects for creating assets.

Excluding *Picture* slots, Type Mismatch is actually the most common error, and this actually increases for *Game* slots. This can be explained by the fact that before *Game* slots, children are only using numeric, string, and list types, as well as `Stencils` and `Shapes`. `Shapes` are created by applying fill or outline functions to `Stencils`, and forgetting to do this before applying geometric transformations would result in a type error. But these type errors are quickly learned, whereas the use of user-defined types in *Game* slots creates many more possibilities for type errors, and even students who understand the mechanisms often make changes to their message or model types and let the compiler show them which parts of the code need to be adapted, because this strategy works well in functional languages like Elm.

Most of the remaining errors are easily understood, and follow variations of these patterns of occurence, except for Naming Error, which is caused by misspelling a defined name, incluing type, function and variable names. It is not unexpected that these errors rise in *Game* slots where user-defined types, wider range of functions, and definitions (of characters or background elements) are often used.

It turns out that the error types encountered are far from equally distributed, and being prepared for this and help mentors prepare explanations for the most common errors. Moreover, the most common errors change from slot type to slot type, and there is an easy-to-understand story we can give to mentors in training to both prepare them both for their expected role in fixing errors, but also in knowing that their students will rapidly improve as the progress through the initial lessons.

5.3. Mentor dashboard

Figure 4 outlines a prototype design of a mentor dashboard. This prototype was used so mentors could see historical compilation data play out in the dashboard, so they could gauge the value of different

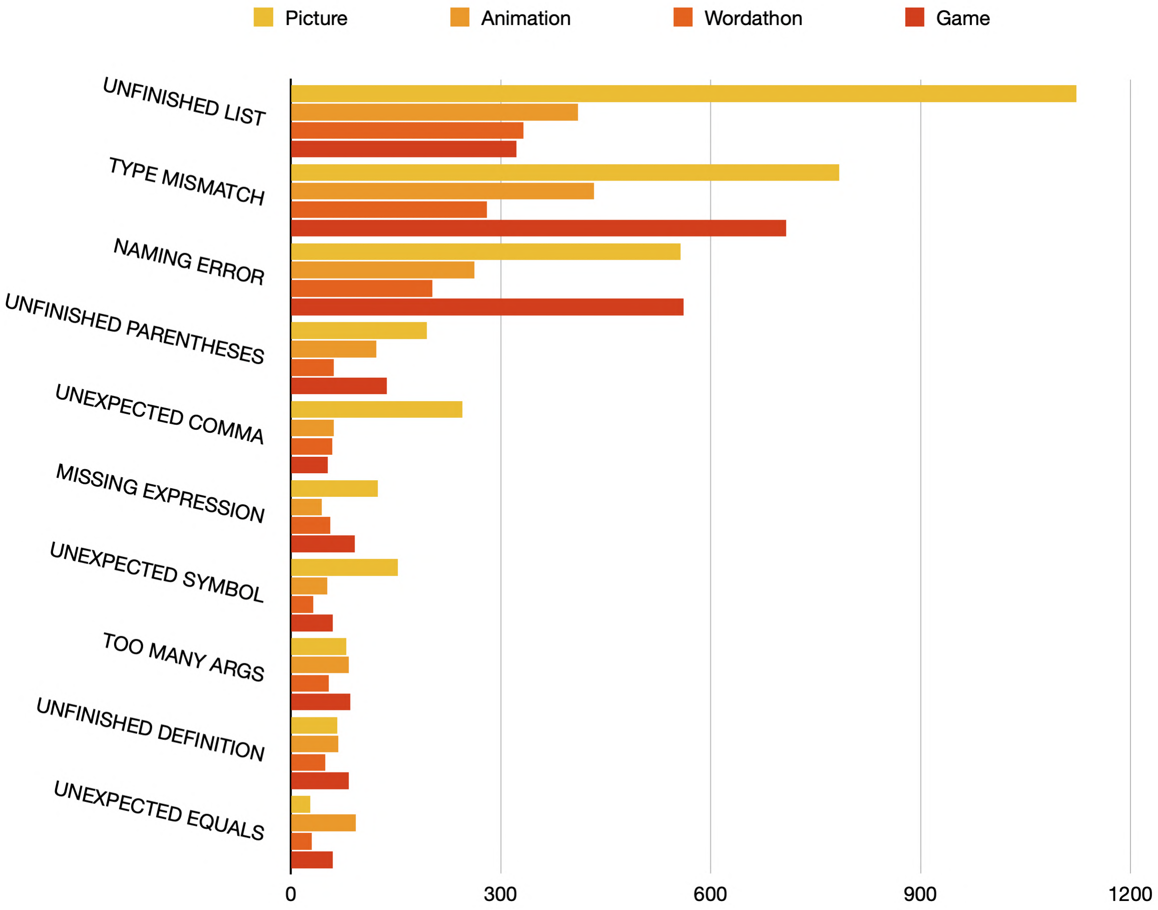


Figure 2: Top 10 errors encountered by users in all slots. Height is the number of compilations resulting in an error.

possible interventions. Although we ultimately did not find useful clusters, the interface was able to display real-time clustering information (see [13, 14]), as well as simple error age (time since the last successful compile) in the upper pane, and per-student aggregate compilation data, in the lower pane. The aging information was considered the most important by mentors, and was integrated into the production mentor dashboard. Clustering and student aggregate data were not integrated, but remain an options for the future.

6. Implications of answers to research questions

Knowing that syntax and type errors are so skewed made it easier to train mentors. If mentors are trained to efficiently solve common errors, they will have more time to handle less common errors. Furthermore, teaching students to avoid the most common errors upfront will reduce the frequency of those errors. Most errors are associated with the Picture activity, and the most common and fourth-most common errors are both related to the construction of lists. We know from teachers' andecdotal reports that children do not study English grammar as a separate subject, and most do not correctly punctuate sentences. It is not surprising that they make many errors in constructing lists. In addition to teacher training, we have developed instructional aids to help mentors prepare students for these challenges. Figure 5 shows one such example. Children between ages 10 and 12 are most likely to make grammar mistakes in English because they are the youngest students we regularly teach. For the same reason, children most remember building things with blocks, so we developed a graphical analogy between the structure of a list and the structure of a house made from blocks. Lacking a broad base, the house is

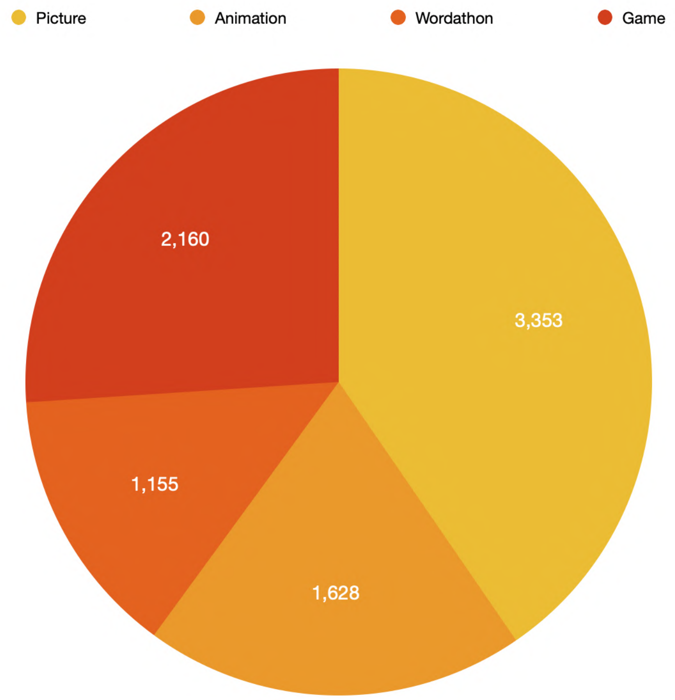
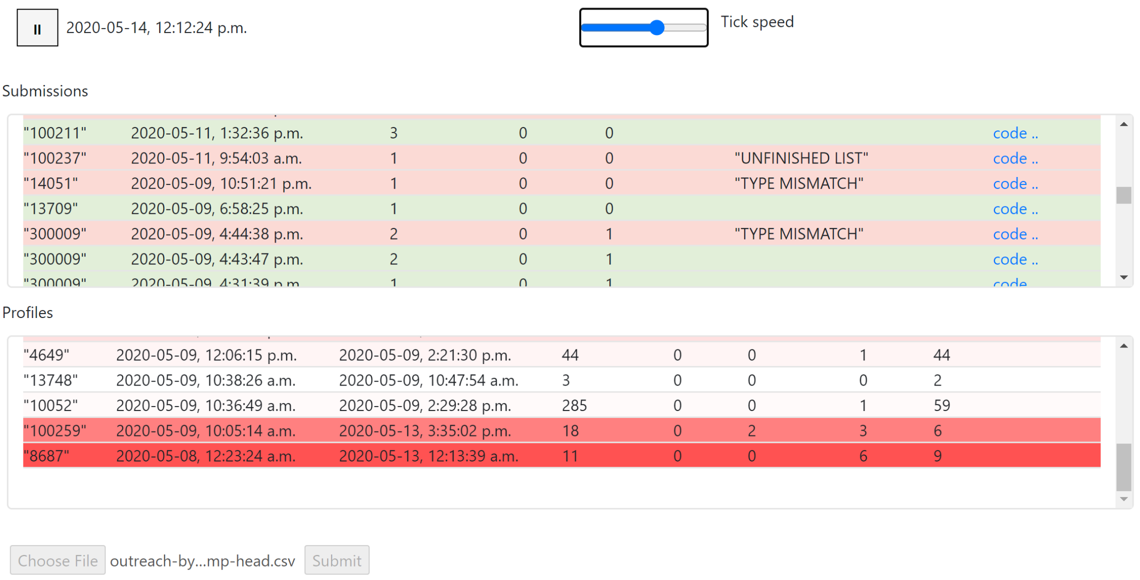


Figure 3: Number of compiles resulting in an error of each type.



2020-05-14, 12:12:24 p.m.

Submissions

"100211"	2020-05-11, 1:32:36 p.m.	3	0	0		code ..
"100237"	2020-05-11, 9:54:03 a.m.	1	0	0	"UNFINISHED LIST"	code ..
"14051"	2020-05-09, 10:51:21 p.m.	1	0	0	"TYPE MISMATCH"	code ..
"13709"	2020-05-09, 6:58:25 p.m.	1	0	0		code ..
"300009"	2020-05-09, 4:44:38 p.m.	2	0	1	"TYPE MISMATCH"	code ..
"300009"	2020-05-09, 4:43:47 p.m.	2	0	1		code ..
"300009"	2020-05-09, 4:31:39 p.m.	1	0	1		code

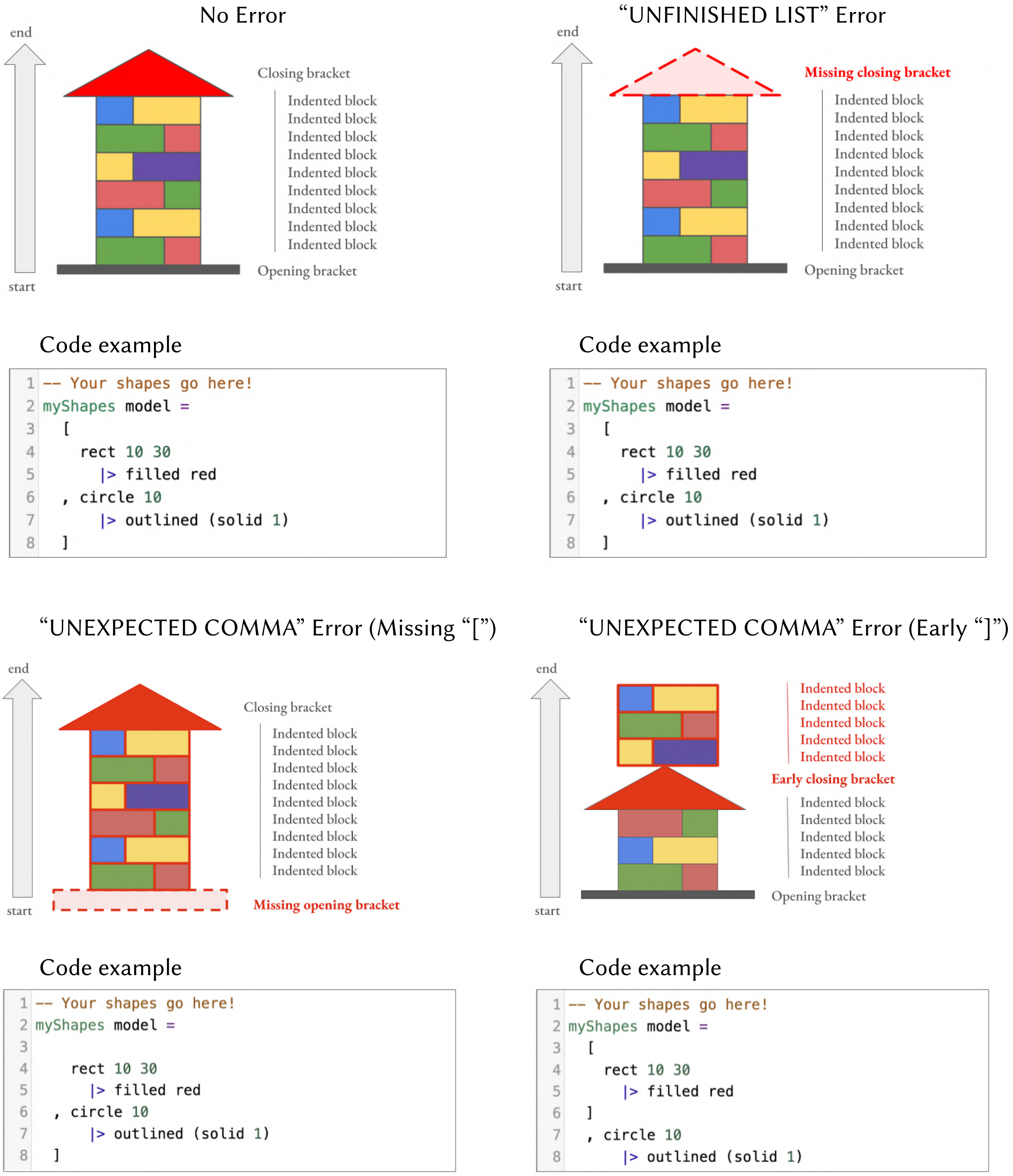
Profiles

"4649"	2020-05-09, 12:06:15 p.m.	2020-05-09, 2:21:30 p.m.	44	0	0	1	44
"13748"	2020-05-09, 10:38:26 a.m.	2020-05-09, 10:47:54 a.m.	3	0	0	0	2
"10052"	2020-05-09, 10:36:49 a.m.	2020-05-09, 2:29:28 p.m.	285	0	0	1	59
"100259"	2020-05-09, 10:05:14 a.m.	2020-05-13, 3:35:02 p.m.	18	0	2	3	6
"8687"	2020-05-08, 12:23:24 a.m.	2020-05-13, 12:13:39 a.m.	11	0	0	6	9

outreach-by...mp-head.csv

Figure 4: Screenshot of a prototype dashboard that mentors could use to gain real-time insights about students. The "Submission" section presents real-time compiles, code insights, and the compilation status. The "Profile" section uses aggregate statistics to inform the mentors of a student's compilation history; the more red a student's profile is, the more unsuccessful compiles they've had. The "Tick Speed" and pause controls are used to control playback of the historical data on compile errors, and would not be part of the real dashboard.

more likely to collapse, analogous to missing the opening bracket. Lacking roof, or having a peaked roof in a middle layer will also cause problems, analogous to missing a closing bracket, or to pasting additional shapes functions after the list has already been closed. Figure 6 shows a second analogy to explain Type Mismatch errors. One of the advantages of strict typing in Elm is that functions can only be composed in meaningful ways, just as blocks can only be connected if they are designed to fit



No Error

Code example

```
-- Your shapes go here!
myShapes model =
  [
    rect 10 30
      |> filled red
  , circle 10
      |> outlined (solid 1)
  ]
```

“UNFINISHED LIST” Error

Code example

```
-- Your shapes go here!
myShapes model =
  [
    rect 10 30
      |> filled red
  , circle 10
      |> outlined (solid 1)
  ]
```

“UNEXPECTED COMMA” Error (Missing “[”)

Code example

```
-- Your shapes go here!
myShapes model =

    rect 10 30
      |> filled red
  , circle 10
      |> outlined (solid 1)
  ]
```

“UNEXPECTED COMMA” Error (Early “]”)

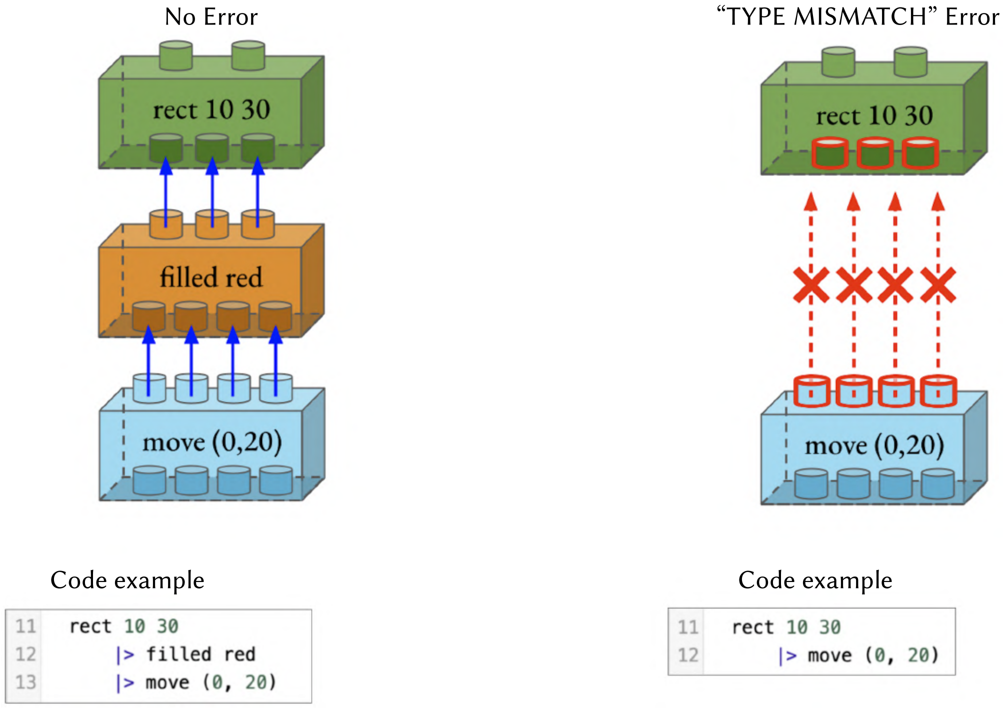
Code example

```
-- Your shapes go here!
myShapes model =
  [
    rect 10 30
      |> filled red
  ]
  , circle 10
      |> outlined (solid 1)
```

Figure 5: Slides using in teaching to make an analogy between list construction and house construction. A well constructed house needs a base to sit on, bricks lined up vertically and a roof. These correspond to the opening `[`, shapes separated by commas, and a closing `]`. Having this analogy seems to help young learners remember to surround lists with brackets, separate elements with commas, and to use consistent indentation.

together. Further study is needed to determine how well such analogies work, and why they work.

As to the second research question. Unfortunately, the specific power law which fit the data indicates that if an error is not solved in less than a minute, it is unlikely to be solved without some intervention. Additional research to elucidate the “why” might lead to other interventions, but for now, our approach is to prioritize individual assistance, and to identify students who have solved errors and are available to help other students. We continue to experiment with dashboards for mentors, instructors and classroom teachers. A dashboard with near real-time previews of student progress allows teachers to identify students who are ready to help others diagnose compiler errors, etc. In Figure 7, we show one such experiment, which combines previews of student work (in this case on a maze challenge) with some



```
rect 10 30
    |> filled red
    |> move (0, 20)
```

```
rect 10 30
    |> move (0, 20)
```

Figure 6: Building block analogy to explain TYPE MISMATCH errors. Functions compose together if their types match, similarly to how blocks snap together if they have the same interfaces.

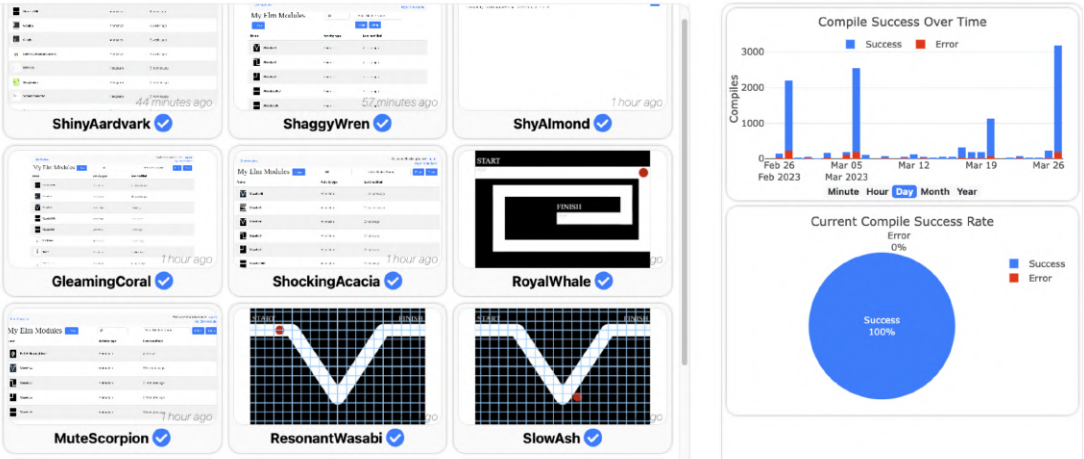


Figure 7: One prototype dashboard helping teachers identify students who are successful at graphical coding tasks, even open-ended creative tasks, and who could be redeployed to help their peers.

overall statistics.

7. Limitations

The programs analyzed for this study were from English-speaking schools in Canada, learning a programming langauge mostly used by web developers, not educators. The results may not generalize.

Our outreach program continues to evolve, and the slot system has been replaced by a more flexible activity system. Our student population is also more diverse, including children with autism and children in enrichment programs. As a consequence, more recent data does not come segmented in the way we relied upon for this analysis. The programs themselves have changed, and we make use of a suite of educational technologies. Beginners can learn about both the basics and some advanced topics outside the WebIDE, which will change the profile and timing of errors.

Despite these factors, the most common errors are largely the same, and the methods developed which are helpful for teaching remain helpful, although the optimal time to introduce them changes.

8. Conclusions and future work

We were able to successfully answer our two research questions: finding that resolution time follows a power law, and hence most errors are resolved relatively quickly, making it easy to decide when to intervene; and developing a story to explain the most common compiler errors and when students are most likely to make them. This information has been incorporated into our curriculum and mentor training.

Author Contributions

Conceptualization – Christopher Kumar Anand; methodology – Yiding Li; formulation of tasks analysis – Kruthiga Karunakaran and Yiding Li; software – Kruthiga Karunakaran and Yiding Li; writing – original draft – Kruthiga Karunakaran and Vaitheeka Nallasamy; analysis of results – Yiding Li and Kruthiga Karunakaran; visualization – Christopher Kumar Anand and Stephanie Li; reviewing and editing – Vaitheeka Nallasamy and Yiding Li. All authors have read and agreed to the published version of the manuscript.

Funding

This study receives funding from the Faculty of Engineering.

Data Availability Statement

No new data were created or analysed during this study. Data sharing is not applicable.

Conflicts of Interest

The authors declare no conflict of interest.

Acknowledgments

We acknowledge financial support from the Faculty of Engineering. We also appreciate input from teachers, parents, and all the enthusiasm and inspiration from all the future coders we visit.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] M. U. Bers, Coding as a literacy for the 21st century, Education Week (2018).

[2] Q. Burke, W. I. O'Byrne, Y. B. Kafai, Computational Participation: Understanding Coding as an Extension of Literacy Instruction, Journal of Adolescent & Adult Literacy 59 (2016) 371–375. doi:10.1002/jaal.496.

[3] D. Silver, M. Saunders, E. Zarate, What factors predict high school graduation in the Los Angeles Unified School District, volume 14, California Dropout Research Project Santa Barbara, CA, 2008.

[4] C. Romero, S. Ventura, Educational data science in massive open online courses, WIREs Data Mining and Knowledge Discovery 7 (2017) e1187. doi:10.1002/widm.1187.

[5] S. Bull, J. Kay, Open Learner Models, in: R. Nkambou, J. Bourdeau, R. Mizoguchi (Eds.), Advances in Intelligent Tutoring Systems, Springer, Berlin, Heidelberg, 2010, pp. 301–322. doi:10.1007/978-3-642-14363-2_15.

[6] W. Matcha, N. A. Uzir, D. Gašević, A. Pardo, A Systematic Review of Empirical Studies on Learning Analytics Dashboards: A Self-Regulated Learning Perspective, IEEE Transactions on Learning Technologies 13 (2020) 226–245. doi:10.1109/TLT.2019.2916802.

[7] J. F. Pane, B. A. Griffin, D. F. McCaffrey, R. Karam, Effectiveness of Cognitive Tutor Algebra I at Scale, Educational Evaluation and Policy Analysis 36 (2014) 127–144.

[8] J. Clarke-Midura, F. Poole, K. Pantic, M. Hamilton, C. Sun, V. Allan, How Near Peer Mentoring Affects Middle School Mentees, in: Proceedings of the 49th ACM Technical Symposium on Computer Science Education, SIGCSE '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 664–669. doi:10.1145/3159450.3159525.

[9] S. Krishnamurthi, K. Fisler, Programming Paradigms and Beyond, in: S. A. Fincher, A. V. Robins (Eds.), The Cambridge Handbook of Computing Education Research, 1 ed., Cambridge University Press, 2019, pp. 377–413. doi:10.1017/9781108654555.014.

[10] E. Czaplicki, S. Chong, Asynchronous functional reactive programming for GUIs, ACM SIGPLAN Notices 48 (2013) 411–422. doi:10.1145/2499370.2462161.

[11] C. d'Alves, T. Bouman, C. Schankula, J. Hogg, L. Noronha, E. Horsman, R. Siddiqui, C. K. Anand, Using Elm to Introduce Algebraic Thinking to K-8 Students, Electronic Proceedings in Theoretical Computer Science 270 (2018) 18–36. doi:10.4204/EPTCS.270.2, arXiv:1805.05125 [cs].

[12] A. Clauset, C. R. Shalizi, M. E. J. Newman, Power-Law Distributions in Empirical Data, SIAM Review 51 (2009) 661–703. doi:10.1137/070710111.

[13] J. MacQueen, Some methods for classification and analysis of multivariate observations, in: Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics, volume 5.1, University of California Press, 1967, pp. 281–298. URL: https://scispace.com/pdf/some-methods-for-classification-and-analysis-of-multivariate-4pswti19oz.pdf.

[14] A. Bhaskara, A. K. Ruwanpathirana, Robust Algorithms for Online k-means Clustering, in: Proceedings of the 31st International Conference on Algorithmic Learning Theory, PMLR, 2020, pp. 148–173. URL: https://proceedings.mlr.press/v117/bhaskara20a.html, iSSN: 2640-3498.