Informatics in Education, 2024, Vol. 23, No. 4, 723–734
© 2024 *Vilnius University*
DOI: https://doi.org/10.15388/infedu.2024.24

Functional Data Science for Secondary-School Students

Paul BIBERSTEIN[1], Thomas CASTLEMAN[2], Luming CHEN[3], Shriram KRISHNAMURTHI[4]
[1] *University of Pennsylvania, USA*
[2] *Aurora Innovation, USA*
[3] *Mathworks, USA*
[4] *Computer Science Department, Brown University, USA*
e-mail: paulbib@seas.upenn.edu, thomas_castleman@alumni.brown.edu, luming_jason_chen@alumni.brown.edu, shriram@brown.edu

Abstract. CODAP is a widely-used programming environment for secondary school data science. Its direct-manipulation–based design offers many advantages to learners, especially younger students. Unfortunately, these same advantages can become a liability when it comes to repeating operations consistently, replaying operations (for reproducibility), and also for learning abstraction.

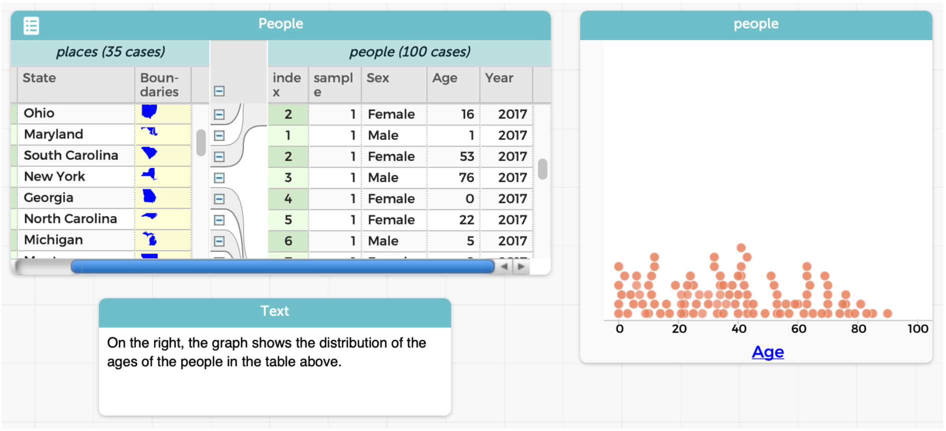
In response, we have extended CODAP with CODAP Transformers, which add a notion of functions to CODAP. These provide a gentle introduction to reuse and abstraction in the data science context. We present a critique of CODAP that justifies our extension, describe the extension, and showcase some novel operations. Our extension has been integrated into the CODAP codebase, and is now part of the standard CODAP tool. It is already in use by the Bootstrap curriculum.

Key words: data science, functional programming, CODAP.

1. Data Science in Schools

Data science curricula are increasingly popular at the secondary school level. These often start from the US 6th grade (roughly age 12) onward, though some curricula aim for even younger ages. At this level, the focus is naturally not on high degrees of technical sophistication. Curricula instead want to give students a sense of data literacy with small amounts of computing: that there are many, rich data sources; that we can process these data using computation; that these computations help us learn facts about the world from these data; and that these data encode various norms and may hide important information.

Our work is situated in the context of a curriculum family called Bootstrap (`https://bootstrapworld.org`), specifically the Bootstrap:Data Science (`https://bootstrapworld.org/materials/data-science`) content. Earlier papers (Krishnamurthi *et al.*, 2019; Schanzer *et al.*, 2022) describe the goals and content of the curriculum in some detail, and provide a preliminary evaluation. The programming described there is done using Pyret (`https://pyret.org`), a student-friendly programming lan-


People

places (35 cases)		people (100 cases)				
State	Boundaries	index	sample	Sex	Age	Year
Ohio		2	1	Female	16	2017
Maryland		1	1	Male	1	2017
South Carolina		2	1	Female	53	2017
New York		3	1	Male	76	2017
Georgia		4	1	Female	0	2017
North Carolina		5	1	Female	22	2017
Michigan		6	1	Male	5	2017

Text

On the right, the graph shows the distribution of the ages of the people in the table above.

Fig. 1. A CODAP workspace containing a table, a graph, and some text

guage based on languages like Python, OCaml, and Racket, but with many of the warts and rough edges removed.

Unfortunately, Pyret programming can be a heavy lift for younger students. They have to learn a specific syntax, deal with error messages (no matter how much work has gone into their design (Wrenn and Krishnamurthi, 2017)), and learn commands to import, process, and visualize data. Furthermore, they cannot perform these operations (such as build visualizations) via direct manipulation, which can sometimes be simpler and more direct.

A common response to these constraints is to use a block-based language such as Scratch (Resnick *et al.*, 2009). However, this only addresses the syntax issues; the direct-manipulation is limited to *constructing programs*, not to *performing programmatic operations*. Therefore, we instead chose to build atop CODAP, which we describe in Section 2. Then, in Section 3, we describe CODAP's weaknesses from the perspective of our curricular needs. The core of the paper, Section 4, describes our innovation in this area. Our extension, CODAP Transformers, is already in use in the Bootstrap:Data Science curriculum (Bootstrap, 2022). The materials are available completely for free on-line, so we encourage readers to examine and use them!

2. An Appreciation of CODAP

CODAP, created by The Concord Consortium (2014) and shown in Figure 1, is a widely-used tool for teaching data science at the secondary school level (National Academies of Sciences, Engineering, and Medicine, 2023). The sources of its appeal are clear:

- Its basic datatype is the table, which researchers have shown even quite young students can work with very comfortably (Konold *et al.*, 2014).
- The tables are enriched with spreadsheet-type formulae.

- Cases are connected between different displays so that, for instance, clicking on a value in a table highlights the corresponding entries in derived graphs, while clicking on points in graphs highlights the original tabular entries.
- Unlike a spreadsheet, multiple objects—tables, graphs, etc.—can coexist independently on screen, arrayed atop a desktop-type metaphor.
- Many operations are performed through direct manipulation, which is probably simpler to learn than textual programming for many users, especially younger users and beginners.

CODAP is also a robust tool that runs entirely within the browser, thereby avoiding software installation issues and enabling easy sharing.

3. A Critique of CODAP

Unfortunately, CODAP also has weaknesses, some of which are tied to the very strengths we describe above. We illustrate these with an example. To be clear, these are not "bugs" in CODAP; the tool is behaving essentially as intended. However, the consequences point to important differences of opinion in goals and ease-of-use.

Let's say we start with the population dataset in Figure 1 and want to find all the people who are below 50 years of age. In CODAP, there are a few ways to do this, all of which have their issues.

If you don't mind being imprecise (which may be okay for a quick data exploration, but isn't if you want to, say, compute a statistic over the result):

- Create a new graph.
- Drag the `Age` attribute ("column") to the graph.
- Select all the cases ("rows") that are under 50 using visual inspection. (Depending on how much data you have and their spread, you'll quite possibly under- and/or overshoot.)
- Then do the last few steps below.

If instead you care to get an accurate selection, begin with:

- Add a new attribute to the original table.
- Enter a formula for that attribute (in this case, `Age < 50`).
- Obtain a selection of the desired cases, which can be done in several different ways, also all with trade-offs:

    1. Sort by that attribute. Unfortunately, this won't work if there's grouping in the table. You'd have to select manually. (We encourage the reader to try this out: it may be a bit harder than it seems.)
    2. Create a graph as above, but of the *new* attribute. This will give you a clean separation into two values. Manually select all the values in the `true` attribute. At least now it will be visually clear if you didn't select all the right values (assuming that the dataset is not too large!).

3. Remove the formula for the new attribute. Now drag it to the leftmost end of the table. (If you don't remove the formula, you'll get an error!) Now you have all the elements grouped by `true` and `false` (and operations performed to one can also be performed to the other).

Either way, you're not done! You still have more steps to go:

- If you aren't already in the table (e.g., if you made a graph), select the table.
- Click on the "Eye" icon.
- Choose the "Set Aside Unselected Cases" entry.

Note that, in most or all of these cases:

- You've added a completely superfluous attribute to your dataset.
- You may have changed the order of cases in your dataset.
- You've lost the ability to see the original data alongside the filtered data.
- You had to take numerous steps.
- You had to remember to use the Eye icon for filtering, as opposed to other GUI operations for other tasks.
- You had to remember where the Eye icon even *is*: it's hidden when a table isn't selected.

But most of all, in every single case:

- You had to perform all these operations **manually**.

Why does this matter? We need data science to be reproducible: we should be able to give others our datasets and scripts so they can re-run them to check that they get the same answer, tweak them so they can check the robustness of our answers, and so on. But when all the operations are done *manually*, there's no "script," only output. That focuses on answers rather than processes, and is anti-reproducibility.

In contrast, we think of filtering as a *program operation* that we apply to a table to produce a new table, leaving the original intact: e.g., the way it works in Pyret. This addresses almost all of the issues above.

Other Pedagogic Consequences. CODAP had to make certain design choices. They made good choices for some settings: for younger children, in particular, the direct manipulation interface works very nicely. It's a low floor. However, we feel it's also a lower-than-we'd-like ceiling. There are many things that the CODAP view of data transformation inhibits:

- Making operations explicit, as we noted above.
- Introducing the idea of **functions** or **transformations** of data as objects in their own right, not only as manual operations.
- Connecting to related subjects, like algebra, that introduce and highlight functions.
- Saving and naming repeated operations, to learn a bottom-up process of developing abstractions and modules.
- Examining old and new tables side-by-side.

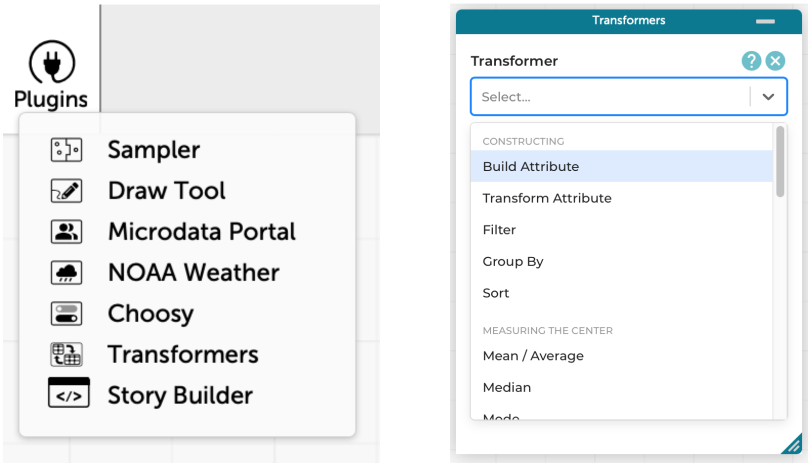


Fig. 2. Left: selecting the Transformers plugin in CODAP. Right: the list of available Transformers.

This last point is especially important. A critical task in data science is performing "what-if" analyses. What-if fundamentally means we should be able to perform some operation (the "if" part) and *compare* the output (the "what" part). We might even want to look at multiple different scenarios, representing different possible outcomes. But traditional what-if analysis, whether in CODAP or on spreadsheets, often requires you, the human, to remember what has changed, rather than letting the computer do it for you.[1]

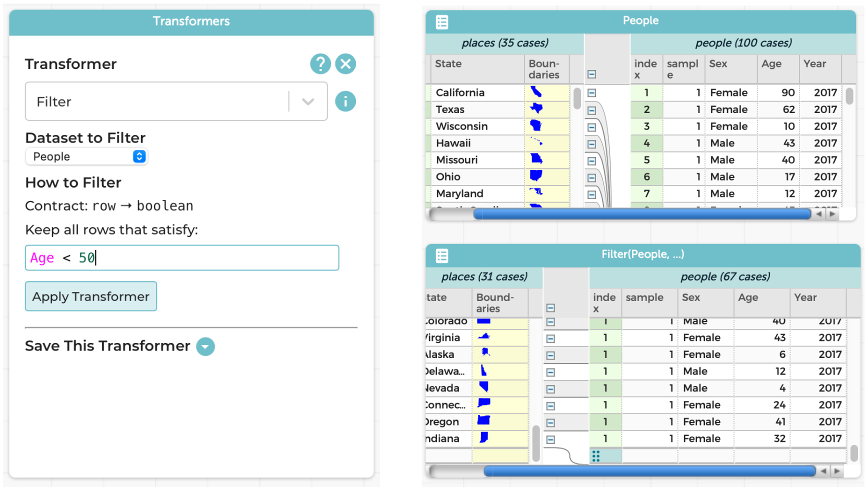
Finally, there's also a subtle consequence to CODAP's design: derived tables must look substantially similar to their parents. In computing terms, the *schema* should be largely the same. That works fine when an operation has little impact on the schema: filtering doesn't change the schema at all (in principle, though in CODAP you have to add an extra attribute...), and adding a new attribute is a conservative extension. But what if you want to perform an operation that results in a radically different schema? For instance, consider the "pivot wider" and "pivot longer" operations when we create tidy data (Wickham, 2014). The results of those operations have *substantially* different schemata!

4. CODAP Transformers

In response to this critique, we've added a new plugin to CODAP called Transformers. This introduces a new pane that lists several transformation operations, grouped by functionality (see Figure 2).

For instance, with no more textual programming than before (the formula is the same), we can perform the same task as before, i.e., finding all the people younger than 50. The result is a **new** table, which co-exists with the original (see Figure 3).

[1]Spreadsheets like Microsoft Excel now provide explicit tools for what-if analysis (Microsoft Inc., 2024), but these are very limited and still require significant human effort.



```
Age < 50
```

People

places (35 cases)		people (100 cases)				
State	Boun-daries	index	sample	Sex	Age	Year
California		1	1	Female	90	2017
Texas		2	1	Female	62	2017
Wisconsin		3	1	Female	10	2017
Hawaii		4	1	Male	43	2017
Missouri		5	1	Male	40	2017
Ohio		6	1	Male	17	2017
Maryland		7	1	Male	12	2017

Filter(People, ...)

places (31 cases)		people (67 cases)				
tate	Bound-aries	index	sample	Sex	Age	Year
olorado		1	1	Male	40	2017
irginia		1	1	Female	43	2017
laska		1	1	Female	6	2017
elawa...		1	1	Male	12	2017
evada		1	1	Male	4	2017
onnec...		1	1	Female	24	2017
regon		1	1	Female	41	2017
ndiana		1	1	Female	32	2017

Fig. 3. Left: the Transformers pane allows applying operations like “Filter.” Right: applying operations leaves the original table untouched (top) while creating a new output table (bottom).

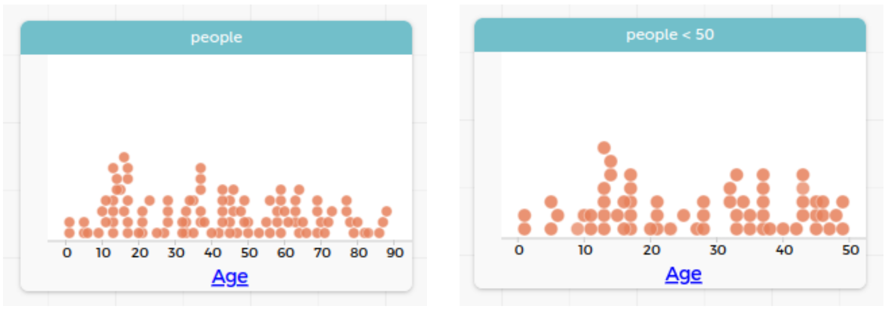
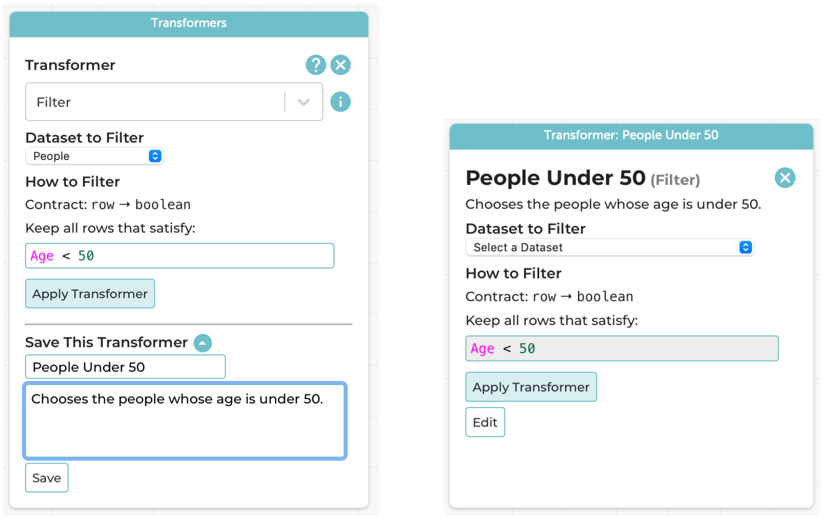


Fig. 4. Left: a graph of `Age` in the original table. Right: a graph of `Age` in the filtered table.

The resulting table is just as much a table as the original. For instance, we can graph the ages in the two tables and see exactly the difference we’d expect (see Figure 4).

Like in the rest of CODAP, the tables built using Transformers follow a dataflow computation process. That is, if the table used as input to a Transformer changes, the Transformer automatically recomputes and propagates changes to the output. If additional Transformers depend on that output, they also update. This means CODAP Transformers users—just like users of spreadsheets—do not need to deal with inconsistent data.

(Over time, of course, the user may build up many tables. The Transformers plugin chooses names based on the operations, to make them easy to tell apart. CODAP also lets you resize, minimize, and delete tables. In practice, we don’t expect users to have more than 3–4 tables up at a time.)


Transformers

Transformer

Filter

Dataset to Filter

People

How to Filter

Contract: `row` → `boolean`

Keep all rows that satisfy:

```
Age < 50
```

Apply Transformer

Save This Transformer

People Under 50

Chooses the people whose age is under 50.

Save

Transformer: People Under 50

People Under 50 (Filter)

Chooses the people whose age is under 50.

Dataset to Filter

Select a Dataset

How to Filter

Contract: `row` → `boolean`

Keep all rows that satisfy:

```
Age < 50
```

Apply Transformer

Edit

Fig. 5. Left: saving a Transformer asks for the Design Recipe steps. Right: saved Transformers freeze their operation, leaving only the table as a parameter.

4.1. *Saving Transformers for Reuse*

We might want to perform the same operation on multiple tables. This is valuable in several contexts:

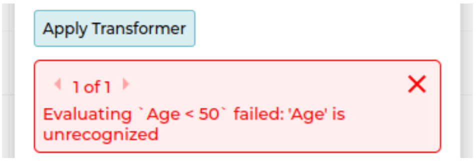
- We create a hand-curated table, with known answers, as a test case to make sure our operations perform what we expect. After confirming this, we want to be sure that we applied exactly the *same* operation to the real dataset.
- We want to perform the same operation to several related datasets: e.g., a table per year.
- We might also simply want to give a meaningful name to the operation.

In such cases, we can use the “Save This Transformer” option at the bottom of the Transformers pane (see Figure 5). Because a saved Transformer is essentially a function, saving follows the Design Recipe of *How to Design Programs* (Felleisen *et al.*, 2018). Thus a saved Transformer has a name, contract, and purpose statement as its interface.

This now creates a new named Transformer (see Figure 5); note that names can be liberal, unlike the rigid rules (like not having spaces) imposed by most textual programming languages. Every part of this new Transformer is frozen other than the choice of dataset; it can be applied as many times as you want, to as many datasets as you want. The above use-cases are suggestions, but you can use it however you wish.

4.2. *A Note on Errors*

Suppose you try to apply an operation improperly. Say, for instance, you have a table of people that does not have an `Age` attribute, and you try to filter people with `Age < 50`. There are at least two choices that Transformers can take:



Evaluating `Age < 50` failed: 'Age' is unrecognized

Fig. 6. Users can attempt to perform operations even if they can be statically checked to be invalid. The result is an informative error message.

1. Allow you to try to perform the operation, and report an error.
2. Prevent you from even trying by simply not showing tables that are invalid in the drop-down list of tables that the operation can be applied to.

The sophisticated programming language, or user interface, designer knows exactly what should happen here: the latter. There is even a design principle that indicates this is what we should do: *make invalid states unrepresentable* (Minksy and Weeks, 2008).

That is not what we chose to do! We instead chose the first option. Here's why.

Imagine you're a teacher with a classroom full of students. A student tries to apply an operation to the wrong table. They probably don't even *realize* that the operation can't be applied. All they know is that the table doesn't appear in the list. *Their table doesn't appear in the list!* Their reaction is (perhaps rightly) going to be to raise their hand and say to their teacher, "This tool is broken! It won't even show me my table!" And the teacher, dealing with a whole bunch of students, all in different states, may not immediately realize why the table doesn't show. Everyone's frustrated; the student feels stuck, and the teacher may be left feeling inadequate.

In contrast, if we just let the operation happen, in our implementation the student sees the error in Figure 6. They now have a pretty good chance of figuring out *for themselves* what went wrong: not pulling away the teacher from helping someone else, not blaming the tool, and instead giving themselves a chance of resolving their situation.

4.3. *Many, Many Transformers!*

We've focused on just one transformation here, but there are many more. We've implemented enough to cover the needs of a data science curriculum, including basic statistical measures, higher order operations inspired by functional programming, and fundamental tabular operations like joins. See them all in Table 1.

4.4. *What-If Analysis*

The Compare Transformer lets users compare numeric and categorical data. These two are handled differently. To illustrate, assume we have the grade book table shown in Figure 7.

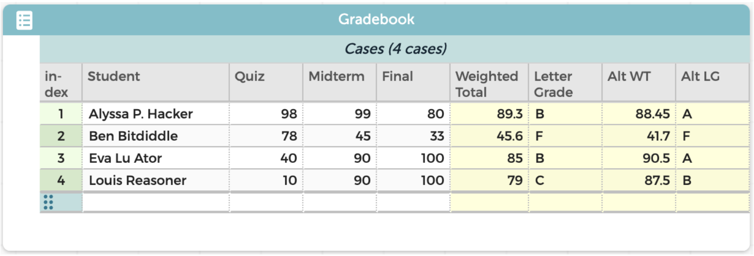
The table shows the grades of students. These have been combined into a weighted average, based on which they have been given a course grade. The instructor has also computed another average using different weights, and different course grades based on a slightly different formula. (The details have been elided because they are not interesting.)

Table 1
A listing of the Transformers available in CODAP Transformers

Operator	Description
Build Attribute	Creates a new attribute based on a formula
Transform Attribute	Modifies an attribute based on a formula
Filter	Filters the table based on a formula
Sort	Sorts the table based on an attribute
Mean/Median/Mode/Standard Deviation	Compute statistical measures over an attribute
Running Sum/Mean/Min/Max	Compute running statistics over an attribute
Difference	Compute the difference between adjacent cases in an attribute
Reduce	Combine a attribute's values using a binary operation
Sum Product	Compute the sum of the element-wise product of two attributes
Count	Count the occurrences of each unique value in an attribute
Compare (Numerical)	Visualize differences between two attributes
Compare (Categorical)	Group by two attributes and merge duplicate cases
Group By	Group by an attribute
Select Attributes	Extract a subset of attributes
Combine Cases	Combine the cases of two tables with identical schemata
Partition	Creates a new filtered table for each unique value in an attribute
Flatten	Remove all group-by operations
Inner Join	Perform an inner join between two tables
Outer Join	Perform an outer join between two tables
Pivot Longer/Wider	Performs the pivoting operations from tidy data
Uneditable Copy	Creates an uneditable copy of a table
Editable Copy	Creates an editable copy of a table that will not receive updates
Copy Structure	Copies the schema from an existing table to a new empty table

Applying the Comparison Transformer in numerical mode to the weighted total attributes produces a new table with two new attributes. One shows the result of numerical subtraction. The other presents a color showing the strength of difference (positive or negative), with the shade proportional to the largest difference of that polarity. Thus the largest negative value will have the strongest red, the largest positive value the strongest green, and all the other values a proportional red or green. Figure 8 shows an example.

While numerical comparison is straightforward, suppose instead we want to compare the resulting final grades. It is perhaps less obvious how to present this. It does not make sense to perform arithmetic on these, but we can compute a different kind of semantic

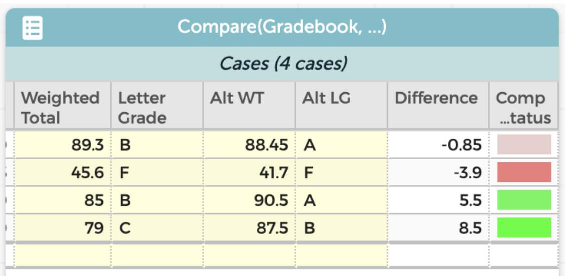


Gradebook

Cases (4 cases)

index	Student	Quiz	Midterm	Final	Weighted Total	Letter Grade	Alt WT	Alt LG
1	Alyssa P. Hacker	98	99	80	89.3	B	88.45	A
2	Ben Bitdiddle	78	45	33	45.6	F	41.7	F
3	Eva Lu Ator	40	90	100	85	B	90.5	A
4	Louis Reasoner	10	90	100	79	C	87.5	B

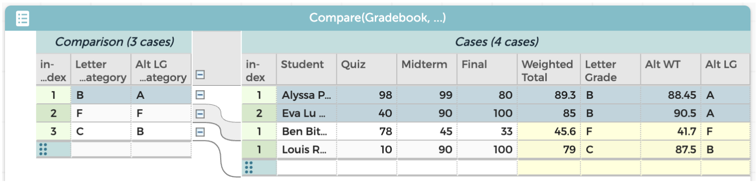
Fig. 7. Sample table for comparisons


Compare(Gradebook, ...)

Cases (4 cases)

Weighted Total	Letter Grade	Alt WT	Alt LG	Difference	Comp ...tatus
89.3	B	88.45	A	-0.85	🟥
45.6	F	41.7	F	-3.9	🟥
85	B	90.5	A	5.5	🟩
79	C	87.5	B	8.5	🟩

Fig. 8. The Numerical Comparison Transformer shows the difference between numerical values in two attributes


Compare(Gradebook, ...)

Comparison (3 cases)

in- ...dex	Letter ...ategory	Alt LG ...ategory
1	B	A
2	F	F
3	C	B

Cases (4 cases)

in- dex	Student	Quiz	Midterm	Final	Weighted Total	Letter Grade	Alt WT	Alt LG
1	Alyssa P...	98	99	80	89.3	B	88.45	A
2	Eva Lu ...	40	90	100	85	B	90.5	A
1	Ben Bit...	78	45	33	45.6	F	41.7	F
1	Louis R...	10	90	100	79	C	87.5	B

Fig. 9. The Categorical Comparison Transformer analyzes the cross-product of two categorical attributes

difference: the cross-product of the categories. Comparison thus produces a new table with cases representing the pairs of grades: A-A (the student would get an A under both calculations), A-B (A in the first calculation and B in the second), A-C, A-F, B-A, and so on. For each such case, there is a *sub-table* of the cases that fit that characterization. This is shown in Figure 9, where we have chosen the B-A case in the Comparison table, which highlights the two corresponding cases of students whose grades would change from B to A. Pairs that have no cases are elided, to keep the Comparison table more manageable.

4.5. *Examples*

Transformers are now part of the official CODAP tool and free for all to use. Here are some pre-built CODAP examples that show the operations in action:

- Building attributes, filtering, and updating data
- Partition
- Filter, running sum, and their interaction
- Transformers that produce single values (as opposed to tables)
- Reusing saved Transformers
- What-if comparisons
- Categorical comparison (alone)
- Numerical comparison (alone)
- Tidy data pivot operations

5. Other Related Work

A recent report (National Academies of Sciences, Engineering, and Medicine, 2023) discusses (§4) tools and resources for data science education in primary and secondary schools. These include unplugged activities, which are outside our scope. The main tools identified were tools for *doing* data science, such as R and Python, and those for *learning* it, of which CODAP was the most prominent example. The report also discusses some plotting and graphing tools, but these are not our focus. In short, as the report makes clear, CODAP Transformers appear to be unique in this space.

Tables are similar to spreadsheets. There have been other approaches to making more flexible spreadsheet-based programming languages, such as Forms/3 (Burnett *et al.*, 2001), but these are not in widespread use in secondary school data science education. On the other hand, Excel has recently adopted higher-order functions (Jones, 2020), but this is a significantly more complex construct than our Transformers.

Finally, the categorical comparison operator in subsection 4.4 is directly inspired by prior research in differential analysis (Nelson *et al.*, 2010; Fisler *et al.*, 2005, 2010).

6. Conclusion

We have presented CODAP Transformers, an extension of the popular CODAP tool used in data science education. CODAP Transformers add a lightweight notion of functions that we believe is consistent with the nature of CODAP (as evidenced by its integration into the official codebase). Transformers allow students to codify computations, name them, and repeat them on multiple datasets. In the process, students learn about functions and responsible data science practices. Transformers have already been integrated into the Bootstrap curriculum. They are also available as Open Source, and are hosted for free on the Web (by the Concord Consortium) for anyone to use.

Acknowledgements

We are deeply grateful to William Finzer, Jonathan Sandoe, and Chad Dorsey (Concord Consortium) for CODAP and for their support. We thank Emmanuel Schanzer, Rachel Tabak, and Flannery Denny (Bootstrap) for valuable conversations and feedback. We also thank Andee Rubin (TERC) for discussing data science and CODAP.

SK is also grateful to Wirth for his pioneering work, especially on Pascal. He vividly remembers the moment when his friend Rustom Guzder told him that Pascal lacked `goto` statements; in response, he asked, “Then how do you program anything interesting in it?” SK eventually got the hang of structured programming. On the other hand, he still revels in continuations and other control operators, which are “`goto`s done right.”

Funding

This work was partially supported by US NSF grants SaTC-2208731 and DRL-2031252.

References

Bootstrap (2022). Bootstrap:Data Science in CODAP. Accessed: 2024-08-31. https://bootstrapworld.org/materials/fall2022/en-us/courses/data-science-codap/index.shtml.

Burnett, M., Atwood, J., Walpole Djang, R., Reichwein, J., Gottfried, H., Yang, S. (2001). Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm. *Journal of Functional Programming*, 11(2), 155–206.

Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S. (2018). *How to Design Programs: An Introduction to Programming and Computing*. The MIT Press. 0262534800.

Fisler, K., Krishnamurthi, S., Dougherty, D.J. (2010). Embracing Policy Engineering. In: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER 2010)*. Association for Computing Machinery, New York, NY, USA, pp. 109–110. 9781450304276. https://doi.org/10.1145/1882362.1882385.

Fisler, K., Krishnamurthi, S., Meyerovich, L.A., Tschantz, M.C. (2005). Verification and Change-Impact Analysis of Access-Control Policies. In: *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pp. 196–205. https://doi.org/10.1109/ICSE.2005.1553562.

Jones, B. (2020). Announcing LAMBDA: Turn Excel Formulas into Custom Functions. https://techcommunity.microsoft.com/t5/excel-blog/announcing-lambda-turn-excel-formulas-into-custom-functions/ba-p/1925546.

Konold, C., Finzer, W., Kreetong, K. (2014). Students' Methods of Recording and Organizing Data. In: *Annual Meeting of the American Educational Research Association*.

Krishnamurthi, S., Schanzer, E., Politz, J.G., Lerner, B.S., Fisler, K., Dooman, S. (2019). Data Science as a Route to AI for Middle- and High-School Students.

Microsoft Inc. (2024). Introduction to What-If Analysis. Accessed: 2024-04-12. https://support.microsoft.com/en-us/office/introduction-to-what-if-analysis-22bffa5f-e891-4acc-bf7a-e4645c446fb4.

Minksy, Y., Weeks, S. (2008). Caml Trading – Experiences with Functional Programming on Wall Street. *Journal of Functional Programming*, 18(4), 553–564. https://doi.org/10.1017/S095679680800676X.

National Academies of Sciences, Engineering, and Medicine (2023). *Foundations of Data Science for Students in Grades K–12: Proceedings of a Workshop*. The National Academies Press, Washington, DC. 978-0-309-69815-3. https://doi.org/10.17226/26852.

Nelson, T., Barratt, C., Dougherty, D.J., Fisler, K., Krishnamurthi, S. (2010). The Margrave Tool for Firewall Analysis. In: *Proceedings of the 24th International Conference on Large Installation System Administration (LISA 2010)*. USENIX Association, USA, pp. 1–8.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., et al.(2009). Scratch: Programming for All. *Communications of the ACM*, 52(11), 60–67.

Schanzer, E., Pfenning, N., Denny, F., Dooman, S., Politz, J.G., Lerner, B.S., Fisler, K., Krishnamurthi, S. (2022). Integrated Data Science for Secondary Schools: Design and Assessment of a Curriculum. In: *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education (SIGCSE 2022)*. Association for Computing Machinery, New York, NY, USA, pp. 22–28.

The Concord Consortium (2014). Common Online Data Analysis Platform, Concord, MA.

Wickham, H. (2014). Tidy Data. *The American Statistician*, 14. https://doi.org/10.18637/jss.v059.i10.

Wrenn, J., Krishnamurthi, S. (2017). Error Messages are Classifiers: A Process to Design and Evaluate Error Messages. In: *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2017)*. Association for Computing Machinery, New York, NY, USA, pp. 134–147. 9781450355308. https://doi.org/10.1145/3133850.3133862.