Types for Tables: A Language Design Benchmark

Kuang-Chen Lu[a], Ben Greenman[a], and Shriram Krishnamurthi[a]

a Brown University, Providence, RI, USA

Abstract

Context Tables are ubiquitous formats for data. Therefore, techniques for writing correct programs over tables, and debugging incorrect ones, are vital. Our specific focus in this paper is on rich types that articulate the properties of tabular operations. We wish to study both their *expressive power* and *diagnostic quality*.

Inquiry There is no "standard library" of table operations. As a result, every paper (and project) is free to use its own (sub)set of operations. This makes artifacts very difficult to compare, and it can be hard to tell whether omitted operations were left out by oversight or because they cannot actually be expressed. Furthermore, virtually no papers discuss the quality of type error feedback.

Approach We combed through several existing languages and libraries to create a "standard library" of table operations. Each entry is accompanied by a detailed specification of its "type," expressed independent of (and hence not constrained by) any type language. We also studied and categorized a corpus of (student) program edits that resulted in table-related errors. We used this to generate a suite of erroneous programs. Finally, we adapted the concept of a datasheet to facilitate comparisons of different implementations.

Knowledge Our benchmark creates a common ground to frame work in this area. Language designers who claim to support typed programming over tables have a clear suite against which to demonstrate their system's expressive power. Our family of errors also gives them a chance to demonstrate the quality of feedback. Researchers who improve one aspect—especially error reporting—without changing the other can demonstrate their improvement, as can those who engage in trade-offs between the two. The net result should be much better science in both expressiveness and diagnostics. We also introduce a datasheet format for presenting this knowledge in a methodical way.

Grounding We have generated our benchmark from real languages, libraries, and programs, as well as personal experience conducting and teaching data science. We have drawn on experience in engineering and, more recently, in data science to generate the datasheet.

Importance Claims about type support for tabular programming are hard to evaluate. However, tabular programming is ubiquitous, and the expressive power of type systems keeps growing. Our benchmark and datasheet can help lead to more orderly science. It also benefits programmers trying to choose a language.

ACM CCS 2012

- **Software and its engineering → Software notations and tools;**
- **Information systems → Query languages;**

Keywords type systems, tabular programming, data science, benchmarking

The Art, Science, and Engineering of Programming

Perspective The Empirical Science of Programming

Area of Submission Database programming, General-purpose programming, Program verification, Programming education

1 Motivation

Tables are a widely-used means of communicating information. They suggest a clean and useful visual representation, and they save data-processors from parsing. They are readily understood and created even by children [36]. Thus, it is unsurprising that a large quantity of data—e.g., government data repositories about everything from demographics to voting to income—are provided as tables (often as CSV files). Furthermore, many datasets are provided in siblings of tables such as spreadsheets and relational databases.

In turn, many programming languages support tabular programming. Some, like SQL, are custom languages, but for many programmers, it is convenient (especially when tables are of moderate size, so that performance is less of a concern) to process tables from within whatever language they are using to write a larger application, or with which they are already very comfortable.

Tables are a rich source of typing discipline. Typically, each column of a table is homogeneous, but the columns can themselves vary in type. There can also be a large number of columns. This makes the typing of tables interesting. Is it just `Table`? That provides no information about values extracted from a table. Is it `Table<T>`? That implies all data in the table are homogeneous, which is rarely the case. Is `Table` a constructor with a type per column? Given that tables do not have a fixed number of columns, this requires variable arity for constructors. Columns are usually accessible by name. They are often also ordered. And so on (section 3).

Owing to this richness and complexity, several authors have created sophisticated systems to type tables (usually) in higher-order, functional programming languages (section 10). Furthermore, tables offer an opportunity for authors of new, richly-typed languages to showcase what their language can do. We are especially interested in these typing schemes because we are currently designing a typed table library for the Pyret language. If there are known techniques that could meet our needs, we would be delighted to use them.

Unfortunately, there is a significant difficulty in performing scientific comparisons among table types, which we saw first-hand. In Spring 2021, the authors ran a graduate seminar to study the state of the art in rich type systems to support programming with tables.[1] Our focus was on both expressiveness and human-factors—the latter (e.g., error quality) often being the victim of enrichments to the former.

We were left deeply frustrated for two reasons. First, we saw little discussion of human-factors in most of the papers. Second, and even more notably, it proved very difficult to compare the many papers we read. There are many operations over tables, but most papers discussed only a very small, select set of them. Were other operations left out due to pure oversight, because of space, or because they were beyond the power of the type system being proposed? Even when operations were present, they were sometimes in a weaker form than one might wish—leaving the same questions

[1] cs.brown.edu/courses/csci2950-x/2021

Figure 1 Live View of Type System Designers Approaching Tabular Programming [54]

unanswered. Ultimately, our efforts to summarize the research landscape (section 10) were curbed by the narrow focus of the papers (informally characterized in figure 1).

We want to be clear that we do not blame any of these authors. Typing tables is a large problem. There are many table processing operations to support. Operations have complex behavior, so their type can be expressed at different levels of richness. Error reporting is another challenge. Each paper has advanced the state of the art by imposing constraints and focusing on part of the problem, which is a reasonable way to make progress.

However, we believe it would be healthy for the community to have a fixed reference point to use as a comparison. Some researchers may be able to show that their systems can already capture all these constraints. Others may be spurred on to new research challenges. Authors of new richly-typed languages can validate their advances over prior work. Researchers who make progress on error reporting can use an objective, third-party suite to demonstrate their improvements. Finally, programmers can use this suite as a guide to understand completeness, richness, and diagnostic aid when choosing languages. Thus, we anticipate salutary outcomes for both the research and programming communities.

While the benchmark provides axes of standardization, it does not automatically dictate how results should be reported. Wide variation in reporting is therefore a risk, and would vastly reduce the value of the benchmark. However, because type systems and programming languages are engineering media, ideas from other forms of engineering may help here. We propose adapting the idea of a datasheet to standardize reporting. In this paper, we introduce a first version of such a datasheet (section 8).

2 Benchmark Components and Design

This paper's contribution is B2T2, the Brown Benchmark for Table Types. The purpose of the benchmark is to serve as a focal point for research on type systems for tabular programming. To this end, we have collected a set of table operations that is sufficient for many tasks and have annotated these operations with precise constraints for type systems to strive for. In addition to this central Table API, the benchmark has four auxiliary components to validate the end-to-end experience of a proposed type system:

1. *What is a Table?* (section 3). A definition of tables. The definition gives candidate type systems a common starting point.
2. *Example Tables* (section 4). A set of example datasets. These are mainly used to illustrate the behavior of operations in other components of B2T2, but also concretize some expressiveness challenges.
3. *Table API* (section 5). An API of tabular operations with detailed type constraints. The constraints are intentionally writted in English to set goals for type system designers in a neutral manner.
4. *Example Programs* (section 6). A set of short programs that use the table operations. Each program raises specific typing questions, such as how to iterate over the numeric columns in a table.
5. *Errors* (section 7). A set of buggy programs with explanations that describe the exact bug in the program. The challenge for a type system is to provide accurate, and ideally actionable, feedback.

To encourage comparisons among implementations of B2T2, the benchmark concludes with a datasheet template for reporting purposes:

6. *Type System Datasheet Template* (section 8). A short free-response form that authors can use to describe their implementation of B2T2.

2.1 Benchmark Source

B2T2 is implemented as a public GitHub repository:

github.com/brownplt/B2T2

Our vision is for the benchmark to serve as a living artifact for the community. First of all, we encourage type system designers to contribute implementations of the benchmark. Second, we invite criticism of the benchmark itself. This initial release may contain implicit assumptions, despite our efforts to weed them out. Similarly, other researchers may propose new elements to enrich the main components—characteristics, example tables, operations, programs, or errors.

2.2 Benchmark Origins

We have constructed B2T2 drawing on several major tabular programming frameworks: R (Tidyverse) [19, 67], Python (pandas) [57], Julia [31], LINQ [43], and SQL [17]. We chose these sources because they are widely regarded as quality data-processing tools and are used in numerous domains in industry and elsewhere. Thus, they provide us with a sense of authenticity.

Readers will notice that the frameworks are mostly from "dynamically typed" languages. This choice is intentional. Starting with a typed library might restrict us to operations that are only expressible in those type systems. In dynamic languages, programmers are not hampered by any particular static type system and can more freely express the behaviors that they find convenient.

B2T2 is meant to be a compelling yet moderately-sized challenge for type system designers. Thus, rather than re-create a full library in the same vein as the inspiring frameworks, we have taken steps to curate an API. In particular, we have:

1. Been selective in copying operations, focusing on a sufficient set of operations to reveal challenges for type systems. For example, the common operation nlargest is missing because it can be encoded with two API operations: tsort and head.
2. Limited the set of parameters and options of the ones we do copy. For instance, B2T2 distills the eight cases of the pandas join operation down to one.
3. Minimized the use of overloading. Instead of including one concat operation that can append rows or columns depending on a parameter, B2T2 contains two operations: vcat and hcat. Section 5 explains the minimal overloading that is included.
4. Translated overly-dynamic features. For instance, some of these libraries let programmers pass expressions in the form of strings. Not only does this depend on an eval-like feature in the language, it also leaves ambiguous how scope is handled. We have instead used function parameters, which avoid all these problems.

In addition to the frameworks, B2T2 is inspired by one more, rather different, source of inputs: Pyret, an educational language, as used in the Bootstrap:Data Science [9] curriculum; and the data-centric [37, 53] collegiate curricula. The use of these curricula serves as a check on the sufficiency of our operation choices; it ensures that the Table API can clearly represent fundamental data-processing tasks.[2]

2.3 Design Alternatives

Before we present the components of B2T2, we pause to discuss a few significant non-goals of the design:

- B2T2 is meant to improve the *normal design* [63] of type systems. Although (aspects of) the benchmark may be useful in other settings—as a reference point for a new domain-specific language, as a source of example programs and errors, etc.—we fully acknowledge that B2T2 is not an appropriate tool to validate *radical* [63] category-breaking methods of tabular programming. Section 9 discusses this limitation in more detail.
- On a similar note, B2T2 is a benchmark for expressiveness aspects of a type system. It is not concerned with the efficiency of operations. Nor does it include broader approaches to evaluation such as cognitive dimensions [8] and conceptual design [28].
- There are several aspects of the programming experience that B2T2 does not cover. Most of all, we mention the ergonomics of the programming interface. Today, one can program using a variety of tools, from structured editors [58] and block-based

[2] Full disclosure: we hope to one day design a richly-typed table library for Pyret.

programming [40] to dot-driven metaphors [48] and new modalities [2, 47, 49]. It has not been clear how to capture these in a technical manner, and some of these anyway stray quite far afield from our focus on types. Section 7 briefly returns to this issue.

3 What is a Table?

github.com/brownplt/B2T2/blob/v1.0/WhatIsATable.md

B2T2 begins with a definition of what we consider to be a table, since the term does not have complete agreement. We intentionally do not over-specify the definition to avoid precluding some clever encoding that we have not envisioned. Rather, we list those characteristics that we consider essential and highlight key design choices.

3.1 Basic Definitions

- A *table* has two parts: a schema and a rectangular collection of cells.
- A *schema* is an ordered sequence of column names and corresponding sorts.
  - The column names must be distinct (no duplicates).
  - The sorts can vary freely.
- A *header* is a sequence of distinct column names (a schema without sorts).
- A *column name* is a string-like first-class datatype.
- A *sort* describes the kind of data that a cell may contain.

Common sorts are numbers and strings; uncommon sorts include images, sequences, and other tables.[3]

- The collection of cells has $C * R$ members, where:
  - C is the length of the schema;
  - R is an arbitrarily-large number of rows; and
  - each cell has a unique index (c, r) for $0 \leq c < C$ and $0 \leq r < R$.

The rectangular arrangement has four important consequences: the rows are ordered, the columns are indexable by schema, all columns contain exactly R cells, and all rows contain exactly C cells.

- A *row* is an ordered sequence of cells.
- A *cell* is a container for data.
  - Cells may be empty.
  - The data in cells of column c must match the sort of the c-th element of the schema.

[3] We use the term "sort" to avoid any intuitive constraints that readers might attach to the term "type". All types are sorts. If needed, a sort can be more.

3.2 Additional Characteristics

- Empty tables, with no cells, may have zero rows and/or zero columns.
- Tables can be represented in row-major order, in column-major order, or in any other way that supports the basic definitions and the Table API operations. Encodings of tables that use other abstractions (functions, objects) are quite welcome as well.
- For this first version of the benchmark, we assume that tables are immutable. Supporting mutation adds significant but largely orthogonal complications: some systems may need to layer on effect systems [38], and all systems have to manage aliasing for soundness. Furthermore, many rich type systems are built atop purely-functional languages. Since one reason to permit mutation is for efficiency, a topic we are explicitly ignoring, this omission does not create problems elsewhere.
- Column sorts are not first class. Nor do we assume any reflective operations on sorts. Consequently, a program cannot filter the numeric columns from a table by inspecting column sorts and the describe/summary functions of R, pandas, and Julia are inexpressible.
- Finally, we ignore input-output: the benchmark does not stipulate how tables are entered into programs. There may be a variety of mechanisms: typed in verbatim, loaded from a file, inserted using a drag-and-drop interface, and so on. We only expect that there be some way to express table constants like those in section 4.

3.3 Typing Tables

Tables have several features that present challenges for conventional type systems, especially because table operations can manipulate aspects of a table. We list these features and their justification below:

- *Columns are heterogeneous*. The column sorts in a table schema allow different kinds of data to sit side-by-side. As a basic example, a table may have numbers in its first column and strings in its second. This property is critical to describe existing datasets, but it does not fit with type systems that require homogeneous collections. Although programmers can create homogeneity by defining an artificial "supertype" that unifies all the actual types contained in a table, this extra step is an imposition that complicates the boundary between datasets and the programming language.
- *Cells may be empty*. Real-world datasets often lack some entries. It is therefore critical that tables can express empty cells. We do not mandate a particular choice, however, because determining how to represent missing values is complex issue that may vary across languages (figure 2), and these debates have an even longer history in databases [13, 15, 45].
- *Rows and columns are ordered*. Rows are ordered so they can be referenced by index; we ignore here performance issues such as random access. Columns are ordered so that users can keep salient columns side-by-side to compare them visually. (Think about the times you've reordered the columns of a spreadsheet to put a pair of interesting columns beside each other.) Tables must preserve this order at least in their presentation, whether or not they do in their internal representation.

	Table	R Tibble	R D.Frame	pandas D.Frame	Julia D.Frame	Julia D.Table[d]
Rectangular	●	●	●	●	●	●
Ordered	●	●	●	●	●	●
Column Names	●	●	●	●	●	●
Distinct Columns	●	●		○	●	●
Row Names			●	●		
Implicit Null	?	●	●		●	

[d] Julia DataTables are deprecated as of May 2021.

Figure 2 The defining characteristics of our tables and a comparison to related work.

- *Column names are first-class and manufacturable*. There are programs for which it is eminently useful to compute the names of columns dynamically. The quizScoreFilter program (section 6) is one example. Of course, such programs are difficult to type because column names are more than atomic labels. Names must be first-class values and require at least append and split operations to build new columns and to compare with existing ones.[4] Other useful operations include pattern-matching on column names, constructing names from other data (e.g. strings and numbers), and building sets of names via unions, intersections, and complements.

3.4 Design Alternatives

Figure 2 presents a more detailed comparison among B2T2 tables, R tibbles (a modern refinement of R data frames [67]), and the data frames found in R, pandas, and Julia. The rows describe notable features. A filled circle indicates the default presence of some feature, a blank space indicates an absence, and an open circle indicates a feature that is configurable but disabled by default. Because B2T2 tables are a specification rather than an implementation, the row for implicit null uses a third symbol ? to mark an unspecified feature.

All designs have *column names* and impose both a *rectangular* shape and *ordered* rows and columns. Designs disagree on the other features: whether columns must be distinct, whether rows have names, and whether there is an implicit notion of null to represent missing data.

- Column names are *distinct* in tables, in tibbles, and in both Julia libraries. The others allow duplicate columns by default and distinguish these columns by position. B2T2 requires distinct columns so that table operations can raise a type error if two columns have the same name. Furthermore, disallowing duplicates may make it easier for SMT-solver-aided type systems to encode schemas.

[4] A language can have first-class names that are not manufacturable; e.g., RASCAL [35].

- Data frames in R and pandas come with *row names*. These names can enable both a handy syntax for accessing data and efficient storage strategies. As a case in point, the F# Deedle library indexes time series data by name rather than position and allows nearest-neighbor lookups.[5] B2T2, by contrast, attaches no metadata to rows. Names must be stored in a column. Incidentally, the tidy data [68] method argues against row names.
- Lastly, the designs are split about whether to encode missing data with an implicit null or not. Tibbles, R data frames, and Julia data frames each come with a sentinel value that may be appear in any column and propagates through common operations. Data frames in pandas do not have a uniform treatment of null; certain Python/pandas types have a null value (e.g. float has np.nan), but other types lack an idiomatic default.[6] Julia data tables do not support null, and instead require the use of an Option type. Interestingly these explicit-null data tables were proposed as an enhancement over Julia data frames, but caused enough breaking changes to warrant a separate package.[7] B2T2 leaves this contentious decision to implementors and merely illustrates situations in which missing data can arise.

4 B2T2: Example Tables

github.com/brownplt/B2T2/blob/v1.0/ExampleTables.md

The second component of B2T2 is a curated set of tables that highlights the basic challenges in representing tabular data. These tables also serve as concrete examples for other parts of the benchmark.

The example tables have the following characteristics:

- They are intentionally small. This is because some languages (especially core languages) may require verbose encodings. There is value to seeing how a small table looks in any system, to compare the systems' ergonomics.
- They contain values that range over a small, but representative set of sorts: numbers, booleans, strings, sequences, and sub-tables.
- Some tables, like gradebookMissing (figure 3), contain empty cells; we indicate these using blank spaces. Each encoding must determine how to handle these.
- The tables jellyAnon and jellyNamed are designed to support operations that iterate over all columns, selecting just those with boolean values.
- The tables employees and departments are designed for use in join operations.

name	age	quiz1	quiz2	midterm	quiz3	quiz4	final
Bob	12	8	9	77	7	9	87
Alice	17	6	8	88		7	85
Eve	13		9	84	8	8	77

Figure 3 Example table gradebookMissing

4.1 A Sample Table

Figure 3 illustrates the gradebookMissing example table. This gradebook resembles data that an instructor might keep. Each row corresponds to a student. The first few columns describe the student. The remaining columns contain numeric grades for different assignments. Note that several column names share a common prefix, quiz, which motivates two table-processing tasks: adding a column (to store the results of a new quiz), and dynamically computing column names (quizScoreSelect in section 6). Two cells are empty, perhaps because the student was absent on a quiz day.

This example table, like others in B2T2, shows only a header instead of a full schema. We let implementations choose appropriate sorts for each column. Of course, implementations are free to require schemas on all table literals.

4.2 Design Notes

The example tables are intended as small illustrations. This narrow focus means that some potential design goals are not met by this first version of the benchmark:

- The example tables are not intended to reflect principles of good test-case design (e.g., various edge cases), and are not meant to form a sufficient test suite.
- These tables also ignore various considerations that arise from programming languages, mathematics, or specific domains. Such considerations include number representations, statistical spread, and serializability.

5 B2T2: Table API

github.com/brownplt/B2T2/blob/v1.0/TableAPI.md

The third and central component of B2T2 is a functional API that supports common data-processing tasks. Each entry in this Table API comes with a conventional sort, a (possibly empty) set of requirements, and a (non-empty) set of guarantees. The challenge for language designers is to express these constraints with types and code.

[5] fslab.org/Deedle/series.html
[6] pandas.pydata.org/pandas-docs/stable/development/roadmap.html#consistent-missing-value-handling
[7] github.com/JuliaData/DataFrames.jl/issues/1148

5.1 Design Goals, Characteristics

- As a collection of operations, the goal of the Table API is to express idiomatic tasks—such as those needed by the curricula noted above (section 2.2)—and to highlight issues for type system design.
  - The API is not meant to be a "core" definition that chooses a minimal set of operations. Rather, the focus is to identify the common tasks for which a core definition should provide a foundation.
  - The API is not meant to serve as a full-fledged table library. For example, it omits a handy subTable operation because that behavior can be expressed as a composition of two included operations: selectColumns and selectRows.
- The requirements and guarantees that annotate each API operation are meant to be complete specifications that describe all properties that a type system might enforce. Additionally, the require/guarantee specifications are written in English to avoid bias toward any particular type system.
- The API includes two pivot operations to support the tidy data style of data cleaning [68]. Notably, these operations show the importance of first-class column names. They require the insertion of column names into table cells and the projection of column names from cells into the schema.
- None of the API operations depend on first-class sorts. Type system designers need not support reflection on types to express the Table API.

5.2 A Sample API Entry

An API entry presents a name, a conventional sort, pre and post conditions, a prose description, and simple examples. Figure 4 sketches the way that these pieces come together for one entry.

5.3 API Format and Conventions

A full description of the API would not make for interesting reading. Therefore, we defer documentation of the full API to the repository link at the top of this section. We do, however, need to explain a few notational conventions in the API that are not intended as constraints on language designers.

1. The Table API splits overloaded operations into separate definitions. The selectRows operation, for example, expects a table and a sequence that describes which rows to extract. Given a sequence of numbers, it selects the corresponding rows. Given a sequence that contains one boolean per row, it selects using the indices of true values in the sequence. Thus, the API has two definitions:

```
(overload 1/2) selectRows :: t1:Table * ns:Seq<Number> -> t2:Table
  ....
(overload 2/2) selectRows :: t1:Table * bs:Seq<Boolean> -> t2:Table
  ....
```

addColumn :: t1:Table * c:ColName * vs:Seq<Value> -> t2:Table

Where t1, c, vs, and t2 name the respective parts of the sort.

Constraints

Requires:

- c is not in header(t1)

  i.e., c must be a fresh column name
- length(vs) is equal to nrows(t1)

  i.e., the sequence of values must have exactly one element per row

Ensures:

- header(t2) is equal to concat(header(t1), [c])
- for all c' in header(t1), schema(t2)[c'] is equal to schema(t1)[c']
- schema(t2)[c] is the sort of elements of vs
- nrows(t2) is equal to nrows(t1)

Description

Consumes a column name and a Seq of values and produces a new Table with the columns of the input Table followed by a column with the given name and values. Note that the length of vs must equal the length of the Table.

```
> hairColor = ["brown", "red", "blonde"]
> addColumn(students, "hair-color", hairColor)
```

name	age	favorite color	hair-color
Bob	12	blue	brown
Alice	17	green	red
Eve	13	red	blonde

Figure 4 Example API entry

Language designers are welcome to handle overloaded operations in any way makes sense in their language: overloading, distinct operations with related names, subclasses, or something else.

2. If an operation can fail, then its result sort is Error<T> for some sort T. Implementors will need to express error terms in an idiomatic manner (perhaps with a tagged message or an integer flag) and may need to adapt the sorts of such operations.
3. The API uses higher-order functions and other forms of abstraction for convenience. Language designers do not need to support exactly these abstractions as long as they can express a similar behavior, perhaps through inlining. For example, buildColumn expects a function that creates a new value from a row, applies this function to each row, and collects a new table column.

   buildColumn :: t1:Table * c:ColName * f:(r:Row -> v:Value) -> t2:Table

   A first-order language might underapproximate this behavior by proscribing a pattern that users can follow after they have defined an f function.

   The orderBy operation presents a more difficult example because conventional types give only a vague impression of its behavior. This operation, which is a combination of the lazy OrderBy and ThenBy methods of LINQ Enumerables [16], uses a sequence of pairs of functions to lexicographically sort a table. Each pair consists of a getKey function and a compare function such that the result sort of getKey matches the input sort of the compare at hand. Different pairs can employ different sorts; e.g., the first compare may expect numbers while the second compare expects strings.

```
orderBy :: t1:Table
           * Seq<Exists K . getKey:(r:Row -> k:K) * compare:(k1:K * k2:K -> Boolean)>
           -> t2:Table
```

One indirect way to express this behavior is to ask for a single getKey function that returns a tuple and a compare function that lexicographically compares the tuples.

4. We acknowledge that the API is written with rather flexible structural typing in mind. Consider the sort for buildColumn above; it assumes that the language can collect a sequence of values into a column and then append that column to widen a table. A language might not support such operations in a fully-generic manner (perhaps to enable Hindley–Milner inference), in which case it is acceptable for a language's buildColumn to ask for a function that computes an entire row. Other operations may require analogous details to explain how pieces fit together. For instance, the sequence argument to orderBy may be easier to express as a heterogeneous tuple.

5.4 Conformance

The overall purpose of the Table API is to set goals for language designers and to enable comparisons among implementation efforts. An explicit non-goal is to constrain the form of tabular languages and type systems. To help clarify this non-goal, we list several points that the API does not mandate.

- The Table API is not a core definition and does not require any particular set of primitives. An implementation may begin with its own core set of primitives and use those to express API operations.
- Similarly, an implementation need not express each API operation as a standalone function. Other ideas include: syntactic sugar (macros), methods, and compositions of other operations. Depending on the choice, an implementation may change the presentation of API entries to match. For example, methods may require different signatures with one fewer parameter.
- Implementations are welcome to choose entirely different *names* for API operations. Though, to aid comparison, it would be helpful to accompany such changes with a map to the names in B2T2.
- Implementations are also welcome to clarify the *sorts* for operations. The unassuming Table sort almost certainly requires parameters. The generic Seq sort may need to be specialized, perhaps to vectors in some cases and to tuples in others. Languages that track nullable values may need more-precise signatures to clarify which arguments can and cannot be null.

- Implementations need not encode all these properties in the pre and post conditions. Our only requirement is that implementations give an explanation of what is and is not (or, cannot be) expressed to enable comparisons.

6 B2T2: Example Programs

github.com/brownplt/B2T2/blob/v1.0/ExamplePrograms.md

B2T2 contains a set of example programs to test how well an implementation of the Table API supports the development of new typed code. Of particular interest is code that uses type system features not found in the API. Each example has two parts: a problem statement and a reference implementation. We list the problem statements below, with emphasis on the type system challenges that each one presents.

dotProduct This function computes the dot product of two columns in a table (otherwise known as SUMPRODUCT in Excel). A type system should ensure that the columns are in the table and that the sorts of these columns describe numbers. A unit checker might also ensure that the numbers have compatible units.

sampleRows This function selects a random sample of a table's rows. Versions of it are found in many popular table libraries. We choose to include it here because it is effectively stateful, which may make it unwieldy or impossible to express in some languages or type systems. Furthermore, randomness is a particular kind of state that is glossed over by some type systems and not by others. The centrality of randomness and sampling in statistical computation makes it important for users of a programming medium to know how randomness, specifically, will be handled.

pHackingHomogeneous This function illustrates the principle of p-hacking using a jellybean dataset inspired by an XKCD cartoon [72]. All columns in the dataset are boolean-valued.

pHackingHeterogeneous This illustrates the same p-hacking principle, but against an initial dataset where *not* all columns are booleans. Thus, a system that can type the previous example cannot necessarily type this one.

quizScoreFilter This example describes a task that many instructors perform at the end of a course: compute the average quiz score for each student in a gradebook. The gradebook contains a mix of numeric and non-numeric fields, and the numbers denote both quiz scores and exam scores. To find the quizzes, this example iterates through all column names and filters the ones that begin with quiz.

quizScoreSelect This example also computes the average quiz scores for a gradebook. It does so by appending the column name quiz to a few integer suffixes and selecting these computed columns from the gradebook.

groupByRetentive This example categorizes rows of an input table into groups based on the values present in a key column. The output table includes the key column. The Table API describes a similar operation; we include it in both places to check that user-defined functions can express detailed type constraints.

```
> pHacking =
  function(t):
    colAcne = getColumn(t, "get acne")
    jellyAnon = dropColumns(t, ["get acne"])
    for c in header(jellyAnon):
      colJB = getColumn(t, c)
      p = fisherTest(colAcne, colJB)
      if p < 0.05:
        println("We found a link between " ++ c ++ " jelly beans and acne (p < 0.05).")
      end
    end
> pHacking(jellyAnon)
```

Figure 5 Example program pHackingHomogeneous

groupBySubtractive This example categorizes rows of an input table into groups based on the values present in a key column. The output table does not include the key column. Like the previous example, its purpose is to test that user-defined code is no less expressive than API code.

6.1 A Sample Program

Figure 5 presents the example code for the pHackingHomogeneous example. It defines a function named pHacking and invokes this function on one of the B2T2 example tables. The function body sketches an implementation using API operations, general-purpose syntax (e.g. loops and conditionals), and one statistical function (fisherTest):

6.2 Conformance

For language implementors, the ground rules for the Table API apply to the example programs. It is not necessary to express each example strictly as a function, nor to follow the code line-by-line. Furthermore, we can imagine that some programs cannot be expressed as written, with a (functional) abstraction, but can be typed if parts of the code are inlined. It may also be necessary to rewrite the programs to reveal some information to the type system. An implementation should document such variances.

7 B2T2: Errors

github.com/brownplt/B2T2/blob/v1.0/Errors.md

For expressive type systems, effective error reporting can be a major challenge. Thus the final component of B2T2 is a suite of erroneous programs with ground-truth explanations. Each program raises two main questions:

name	age	favorite color
String	Number	String
12	Bob	blue
17	Alice	green
13	Eve	red

The rows disagree with the schema on the ordering of the first two columns.

Figure 6 Malformed table constant swappedColumns

1. Does the type system detect the error?
2. If so, how understandable is its explanation?

There is an implicit third question; namely, is the program even expressible? We address this point below (section 7.2) as part of a larger discussion about how to compare error feedback across languages.

All the examples are based on *actual* errors from a log of student programs (submitted anonymously and voluntarily) in an introductory course at Brown University. Our presentation distills the student programs to eliminate unnecessary, confusing, or personally identifying context, rename variables, refer to our sample tables, etc., while leaving the essence of the problem unchanged. Every entry implictly makes assumptions about a student's intent; in some cases, this is difficult to discern just from the erroneous program. In all cases, we therefore studied the edits that the student subsequently made (which typically ended in a corrected version that ran properly) to understand what they meant to write instead of the erroneous program.

Several examples contain an error due to a malformed table constant. Figure 6 presents one example; the data in this table does not match sorts in its schema. These "obvious" mistakes are nevertheless common, and their inclusion gives table-aware languages a chance to showcase their helpful feedback. By contrast, languages that encode tables with a desugaring may struggle to explain such errors in terms of the surface syntax. The benchmark includes several constant errors because we anticipate that a language may give better feedback to some than to others. For example, the output for an empty cell could be very different from that for an empty row. The latter may give a very simple error whereas the former produces an indecipherable one (due to desugaring, etc.). Or it could be the other way around, with a smart error for the empty cell because of contextual heuristics, and an ugly error for the missing row because there is no context.

7.1 A Sample Error

Each error entry contains five parts: the names of any example tables (section 3) that the program refers to, the program's intent, the buggy program, an explanation of why the code is erroneous, and a corrected version of the program. Figure 7 presents an error involving two column names and a boolean operator.

blackAndWhite

Context

jellyAnon

Task

The programmer was asked to build a column that indicates whether "a participant consumed black jelly beans and white ones."

A Buggy Program

```
> eatBlackAndWhite =
   function(r):
      r["black and white"] == true
   end
 > buildColumn(jellyAnon, "eat black and white", eatBlackAndWhite)
```

What is the Bug?

The logical and appears at a wrong place. The task is asking the programmer to write r["black"] and r["white"], but the buggy program accesses the invalid column "black and white" instead.

A Corrected Program

```
> eatBlackAndWhite =
   function(r):
      r["black"] and r["white"]
   end
 > buildColumn(jellyAnon, "eat black and white", eatBlackAndWhite)
```

Figure 7 Example error entry

7.2 Towards Error Evaluation Criteria

Analyzing the quality of feedback is not a science, and requires some interpretation. Several recent surveys have analyzed the quality of error messages, and thus offer suggestions of techniques [6, 7, 60]. Human factors research on warning label design is also relevant [70]. Our own prior work develops a robust rubric for analyzing error quality based on subsequent programmer actions [41], and also develops a "static" criterion that applies precision and recall to evaluating the quality of messages at design-time [71]. All these ingredients may prove useful to compare error feedback.

One important aspect of benchmarking errors is that the language may have means to preclude their construction entirely. In a traditional, textual language, one can write virtually any text string and submit it for analysis. In contrast, structured editors and block-based editors have a critical property: it is impossible to construct a syntactically ill-formed program. We will call these *preventative* programming media. Types can be considered an extension of this: they are a context-sensitive well-formedness check, and can thus be incorporated into a preventative editor. The burden then shifts to a qualitatively different kind of phenomenon: from explaining an error that *has occurred* (which can reference a concrete program) to explaining why a certain program *cannot be built* (which pertains to a set of programs that, by definition, cannot exist).

We note that there can be a subtle interaction between preventative media and rich type systems. Block languages, for instance, work well because there are obvious visual cues showing why one block cannot be placed inside another. However, when type errors become more subtle—and context-sensitive—preventative methods have the potential to baffle programmers much more than an after-the-fact error report might [39, 51, 64]. Therefore, this domain presents an interesting case-study in the creation of richly-typed preventative programming interfaces.

7.3 Conformance

Unlike the Table API, which permits some freedom of implementation, these error benchmarks are sensitive to small changes. For example, in a sophisticated type system, inlining code can significantly change the detection and, even more so, the reporting of an error. Thus, any deviations must be carefully documented and justified.

8 Type System Datasheet Template

github.com/brownplt/B2T2/blob/v1.0/Datasheet.md

Recently, a group of influential data scientists put forward the notion of *datasheets* for datasets [24]. Those authors are directly inspired by a tradition in engineering:

> In the electronics industry, every component, no matter how simple or complex, is accompanied with a datasheet describing its operating characteristics, test results, recommended usage, and other information.

Datasheets enable engineers to quickly compare similar components and choose ones fit for purpose. Noting the importance of datasets and their potential for misuse, the authors present an analogous notion of datasheets for datasets.

Programming languages and type systems would benefit from similar documentation. The goal of such a datasheet is not to preclude innovation or hide novelty or virtues; rather, *on aspects that can be compared*, a datasheet provides a quick, standard way to determine which component can fit a use. Put differently, the benchmark is an attempt to systematize the “input” and the datasheet helps systematically summarize the “output” (i.e., the reporting of the system).

To this end, we accompany B2T2 with a datasheet template for tabular type systems. The authors of a language that implements B2T2 can fill out the datasheet to help would-be readers quickly understand the new language.

9 How Not to Use B2T2

As mentioned above (section 2.3), B2T2 is a tool for the normal design of tabular type systems. We developed the benchmark in response to a lack of focus among related works—there are many type systems that support tabular programming in a conventional language, and yet these designs are extremely difficult to evaluate (section 10). B2T2 sets a common goal for these type system designs. No matter how many constraints a particular type system can express, implementing B2T2 relates the system to the practice of tabular programming as found in widely-used frameworks.

B2T2 is not, however, a suitable goal for all typed tabular languages. Unconventional programming media may struggle to express aspects of the benchmark. For instance, one can imagine a virtual reality-based environment in which users can directly manipulate tables in space; such an environment might have an awkward time with the Table API even if the environment accommodates the example programs. The Subtext language [21], as well as Pyret, synthesize types from example. These media might encode useful constraints without anything resembling traditional code or types. For such media, it makes sense to ignore parts of B2T2. Therefore, B2T2 should never be used to criticize such designs, which may need very different evaluation methods (such as cognitive dimensions [8] and concepts [28]).

In short, B2T2 is a structured and technical evaluation criterion. Although there are many "normal" type systems to which it applies, there may be "slightly abnormal" designs to which it only partially applies and "radical" designs that are out of scope. These radical designs are nevertheless vital for progress in programming media.

10 Related Work

Programming with Tables As noted above (section 2), B2T2 is directly inspired by tabular programming frameworks. The frameworks include R Tidyverse [19, 59], Python pandas [42, 57], Julia [31, 32], LINQ [43], and SQL [17, 18]. Each provides a toolkit for comprehending and manipulating tables. The B2T2 Table API selects vetted operations from these sources.

The Bootstrap:Data Science [9] and data-centric computing [37, 53] curricula provide critical validation. First, the teaching materials present basic data science tasks that should be expressible. Second, learners that have taken these courses graciously supplied the logs that we used to find erroneous programs (section 7).

Types for Tables There is a huge amount of prior work on type systems that support tabular programming in some form or another. Because these works pursue different goals and present examples that vary widely in complexity, a direct comparison is difficult. We hope that B2T2 enables apples-to-apples comparisons in the future. To a first approximation, however, the research targets five application areas:

- *Records and Variants* Any type system that supports polymorphic records can support a kind of tabular programming, e.g. [23, 27, 44, 50, 65, 66], though the details depend on the allowed operations on records. Historically, these systems focus on decidable type inference and disallow first-class labels.
- *Relational Algebra* The authors of LINQ claim that relational algebra is enough to support programming with a variety of data formats, including tables [43].

Several other languages follow this maxim, including Ferry [26], SML# [10, 46], and Ur [12].

- *Array-Oriented Programming* Remora is a typed variant of the J programming language [30, 52]. The multi-dimensional array operations in Remora can likely be specialized to tabular programming. Qube [61] and FISh [29] might be repurposed in a similar manner.
- *Data Exploration* The Gamma (thegamma.net) is an innovative language for data exploration [48]. Programmers import a dataset as an object and type a dot (.) after the dataset name to see a list of analytical operations. Applying an operation computes a new type for the result using a *pivot* type provider [56], which is enabled by an untyped relational algebra engine. The dot-driven programming model is compelling, and we are curious to learn the extent to which it can accommodate other tabular programming idioms.
- *Fancy Types* The designers of advanced type systems occasionally use tabular programming as an application area to demonstrate expressiveness. Examples include the constant-propagating types in CompRDL [33] and the refinement types of Liquid Haskell [62]. Along these lines, we conjecture that the TypeScript keyof operator [20] can support much of the B2T2 Table API.

Spreadsheet Programming Spreadsheets are not tables, but type systems that detect spreadsheet errors might be useful to detect errors in tables [3, 11, 69]. Tabular programming might also benefit from incorporating some of the more-structured elements of spreadsheet programming, such as reactive equations. These would, however, significantly complicate the APIs.

Language Design Benchmarks We are aware of a few other benchmarks for aspects of language design. One is related and complementary to B2T2: an in-progress expressiveness benchmark [14] that compares several languages and styles of programming on table-processing tasks. The representation design benchmarks [73] identify static representation problems for visual programming languages. POPLMark [5] and POPLMark reloaded [1] present problems for proof assistants. Berkeley Motifs [4] set goals for parallel programming models and architectures. The LWC benchmarks [22] address various aspects of language workbenches, from notation to code reuse. Finally, 7GUIs [34] presents seven tasks for GUI toolkits to express succinctly.

11 Conclusion

Ultimately, our goal is to improve the practice of tabular programming via static typing. Rich types have the potential to help all kinds of users; they can offer documentation for learners, performance hints for experts, and feedback for everyone in between. The demand for such tooling is high, and we expect it to grow in the coming years.

B2T2 is a first step toward this long-term goal, designed to focus our own design efforts and to promote scientific discussions with other research teams. It was born of a frustration: trying to reconcile a large number of different papers that each described exciting advances but that were mutually incomparable. We believe the components of the benchmark cover the basics of a quality tabular language. First, we offer a standard definition that conforms well to real-world tables. Second, the example tables concretize the definition. Third, the Table API provides a curated set of operations to strive for. Fourth, the example programs validate the API and illustrate additional typing issues. Fifth, the error illustrations draw attention to diagnostics. Finally, the datasheet template brings these components into a technical summary that is designed to encourage comparisons.

Benchmarks are normative, however, and we acknowledge the threat posed by Goodhart's Law [25]. In brief, the trouble is that "when a measure becomes a target, it ceases to be a good measure" [55]. That said, two points about B2T2 help to lessen its potential of becoming a disconnected measure:

1. We begin with a fairly useful set of operations that represent a foundation for standard table processing. If a language supported only these operations, that would still provide a comfortable programming basis for many situations, especially when general-purpose programming tools are also available.
2. We expect that language designers (and data scientists) will be happy to update our benchmark with examples we have missed, especially—in the spirit of friendly competition—those they support well, or—in the spirit of scientific honesty—those they support poorly.

In sum, we are optimistic that B2T2 will help attract programming language expertise to the central issues for typed tables. Nevertheless, the cautions of section 9 are critical and we would be disappointed if this work were used to squelch innovation.

Benchmark Links

- GitHub release: github.com/brownplt/B2T2/releases/tag/v1.0
- Zenodo archive: doi.org/10.5281/zenodo.5507463
- Extended paper: cs.brown.edu/research/plt/dl/prog2022-b2t2

Acknowledgements We thank: the attendees of our Spring 2021 seminar for thoughtful discussions; Ben Lerner for suggesting the dotProduct example program; Tomas Petricek for broadening our design discussions; the other reviewers for thoughtful feedback; Titus Barik and Brett Becker for references related to error messages; Margaret Burnett, Will Crichton, Paul Khuong, and Cameron Yick for references to language design benchmarks; Ranjit Jhala for help with Liquid Haskell as we prototyped B2T2; and Matthias Felleisen for early criticism that shaped the direction of this work.

This work was partly supported by the US National Science Foundation. This research was also developed with funding from the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL). The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Greenman received support from NSF grant 2030859 to the CRA for the CIFellows project.

A Datasheet

A.1 Reference

Q. What is the URL of the version of the benchmark being used?

Q. On what date was this version of the datasheet last updated?

Q. If you are not using the latest benchmark available on that date, please explain why not.

A.2 Example Tables

Q. Do tables express heterogeneous data, or must data be homogenized?

Q. Do tables capture missing data and, if so, how?

Q. Are mutable tables supported? Are there any limitations?

You may reference, instead of duplicating, the responses to the above questions in answering those below:

Q. Which tables are inexpressible? Why?

Q. Which tables are only partially expressible? Why, and what's missing?

Q. Which tables' expressibility is unknown? Why?

Q. Which tables can be expressed more precisely than in the benchmark? How?

Q. How direct is the mapping from the tables in the benchmark to representations in your system? How complex is the encoding?

A.3 TableAPI

Q. Are there consistent changes made to the way the operations are represented?

Q. Which operations are entirely inexpressible? Why?

Q. Which operations are only partially expressible? Why, and what's missing?

Q. Which operations' expressibility is unknown? Why?

Q. Which operations can be expressed more precisely than in the benchmark? How?

A.4 Example Programs

Q. Which examples are inexpressible? Why?

Q. Which examples' expressibility is unknown? Why?

Q. Which examples, or aspects thereof, can be expressed especially precisely? How?

Q. How direct is the mapping from the pseudocode in the benchmark to representations in your system? How complex is the encoding?

A.5 Errors

There are (at least) two parts to errors: representing the source program that causes the error, and generating output that explains it. The term "error situation" refers to a representation of the cause of the error in the program source.

For each error situation it may be that the language:

- *isn't expressive enough to capture it*
- *can at least partially express the situation*
- *prevents the program from being constructed*

Expressiveness, in turn, can be for multiple artifacts:

- *the buggy versions of the programs*
- *the correct variants of the programs*
- *the type system's representation of the constraints*
- *the type system's reporting of the violation*

Q. Which error situations are known to be inexpressible? Why?

Q. Which error situations are only partially expressible? Why, and what's missing?

Q. Which error situations' expressibility is unknown? Why?

Q. Which error situations can be expressed more precisely than in the benchmark? How?

Q. Which error situations are prevented from being constructed? How?

Q. For each error situation that is at least partially expressible, what is the quality of feedback to the programmer?

Q. For each error situation that is prevented from being constructed, what is the quality of feedback to the programmer?

B Table API Snapshot, Version 1.0

Datasheet above. Rest below.

Note: the latest version may be found at

github.com/brownplt/B2T2

B.1 What is a Table?

B.1.1 Fundamentals

- A *table* has two parts: a schema, and a rectangular collection of cells.
- A *schema* is an ordered sequence of column names and sorts.
  - Column names must be distinct (no duplicates)
- A *column name* is a string-like first-class value
- A *sort* is a specification of the data that a column contains.
  - Each cell in the i-th column must match the i-th sort.

- Cells may be organized into rows or columns. Either way:
  - All rows must have the same length
  - All columns must have the same length
- Cells may contain data or may be missing data

B.1.2 Auxiliaries

- A *header* is an ordered sequence of column names (a schema without sorts)

B.2 Example Tables

This file lists some tables that are either used in other files (e.g. TableAPI and ExampleProgram), or illustrating interesting structural properties (e.g. having some values missing, having lists in cells, and having tables in cells).

B.2.1 students: a simple table with no values missing.

name	age	favorite color
"Bob"	12	"blue"
"Alice"	17	"green"
"Eve"	13	"red"

B.2.2 studentsMissing: a simple table with some values missing.

name	age	favorite color
"Bob"		"blue"
"Alice"	17	"green"
"Eve"	13	

B.2.3 employees: a table that contains employees and their department IDs (source)

Last Name	Department ID
"Rafferty"	31
"Jones"	32
"Heisenberg"	33
"Robinson"	34
"Smith"	34
"Williams"	

B.2.4 departments: a table that contains departments and their IDs (source)

Department ID	Department Name
31	"Sales"
33	"Engineering"
34	"Clerical"
35	"Marketing"

B.2.5 jellyAnon: a jelly bean table that contains only boolean data

get acne	red	black	white	green	yellow	brown	orange	pink	purple
true	false	false	false	true	false	false	true	false	false
true	false	true	false	true	true	false	false	false	false
false	false	false	false	true	false	false	false	true	false
false	false	false	false	false	true	false	false	false	false
false	false	false	false	false	true	false	false	true	false
true	false	true	false	false	false	false	true	true	false
false	false	true	false	false	false	false	false	true	false

```
| true     | false | false | false | false | false  | true  | true   | false | false  |
| true     | false | false | false | false | false  | false | true   | false | false  |
| false    | true  | false | false | false | true   | true  | false  | true  | false  |
```

B.2.6 jellyNamed: a jelly bean table that contains booleans and strings

```
| name       | get acne | red   | black | white | green | yellow | brown | orange | pink  | purple |
| ---------- | -------- | ----- | ----- | ----- | ----- | ------ | ----- | ------ | ----- | ------ |
| "Emily"    | true     | false | false | false | true  | false  | false | true   | false | false  |
| "Jacob"    | true     | false | true  | false | true  | true   | false | false  | false | false  |
| "Emma"     | false    | false | false | false | true  | false  | false | false  | true  | false  |
| "Aidan"    | false    | false | false | false | false | true   | false | false  | false | false  |
| "Madison"  | false    | false | false | false | false | true   | false | false  | true  | false  |
| "Ethan"    | true     | false | true  | false | false | false  | false | true   | true  | false  |
| "Hannah"   | false    | false | true  | false | false | false  | false | false  | true  | false  |
| "Matthew"  | true     | false | false | false | false | false  | true  | true   | false | false  |
| "Hailey"   | true     | false | false | false | false | false  | false | true   | false | false  |
| "Nicholas" | false    | true  | false | false | false | true   | true  | false  | true  | false  |
```

B.2.7 gradebook: a gradebook table with no missing values.

```
| name    | age | quiz1 | quiz2 | midterm | quiz3 | quiz4 | final |
| ------- | --- | ----- | ----- | ------- | ----- | ----- | ----- |
| "Bob"   | 12  | 8     | 9     | 77      | 7     | 9     | 87    |
| "Alice" | 17  | 6     | 8     | 88      | 8     | 7     | 85    |
| "Eve"   | 13  | 7     | 9     | 84      | 8     | 8     | 77    |
```

B.2.8 gradebookMissing: a gradebook table with some missing values.

```
| name    | age | quiz1 | quiz2 | midterm | quiz3 | quiz4 | final |
| ------- | --- | ----- | ----- | ------- | ----- | ----- | ----- |
| "Bob"   | 12  | 8     | 9     | 77      | 7     | 9     | 87    |
| "Alice" | 17  | 6     | 8     | 88      |       | 7     | 85    |
| "Eve"   | 13  |       | 9     | 84      | 8     | 8     | 77    |
```

B.2.9 gradebookSeq: a gradebook table with sequence cells

```
| name    | age | quizzes      | midterm | final |
| ------- | --- | ------------ | ------- | ----- |
| "Bob"   | 12  | [8, 9, 7, 9] | 77      | 87    |
| "Alice" | 17  | [6, 8, 8, 7] | 88      | 85    |
| "Eve"   | 13  | [7, 9, 8, 8] | 84      | 77    |
```

B.2.10 gradebookTable: a gradebook table with table cells

```
| name    | age | quizzes           | midterm | final |
| ------- | --- | ----------------- | ------- | ----- |
| "Bob"   | 12  | | quiz# | grade | | 77      | 87    |
|         |     | | ----- | ----- | |         |       |
|         |     | | 1     | 8     | |         |       |
|         |     | | 2     | 9     | |         |       |
|         |     | | 3     | 7     | |         |       |
|         |     | | 4     | 9     | |         |       |
| "Alice" | 17  | | quiz# | grade | | 88      | 85    |
|         |     | | ----- | ----- | |         |       |
|         |     | | 1     | 6     | |         |       |
|         |     | | 2     | 8     | |         |       |
|         |     | | 3     | 8     | |         |       |
|         |     | | 4     | 7     | |         |       |
| "Eve"   | 13  | | quiz# | grade | | 84      | 77    |
|         |     | | ----- | ----- | |         |       |
|         |     | | 1     | 7     | |         |       |
|         |     | | 2     | 9     | |         |       |
|         |     | | 3     | 8     | |         |       |
|         |     | | 4     | 8     | |         |       |
```

B.3 Table API

This file serves for two purposes:

- Challenge type system designers
- Set up a reference for comparing programming medias on their
  - **expressiveness:** is an operators provided in one media but not the other?
  - **enforcement of constraints:** how many of the required constraints are enforced? How many of the ensured constraints are communicated to the type system?

Real-world programming medias contain lots of operations. Collecting all of them won't be practical or necessary for the purposes of this file. Instead, we strive to gather at least all operators that are necessary for real-world data analysis. (Please let us know if you think a necessary operator is missing.) Furthermore, some operators impose interesting constraints that might be challenging to type systems. We selectively include some of these operators and hopefully they will illustrate all constraints that a type systems need to handle. In short, an operator is included if it meets one of the following criteria:

- necessary for realistic table programming
- illustrating interesting constraints not illustrated by other operators in this file

Operators are collected from the following resources:

- Python pandas
- R dplyr cheatsheets
- R tibbles
- R Tidy data
- Julia DataFrames
- LINQ
- MySQL
- PostgreSQL
- Pyret taught in Brown CS111
- Pyret taught in the Bootstrap DS
  - the definition of methods and some functions
  - the definition of other functions
- Compare Python pandas with R TidyVerse
- Compare Python pandas with SQL
- Compare Julia DataFrame with Python pandas and R TidyVerse

For our convenience, we sometimes apply table operators to rows (e.g. `selectColumns(r, ["foo", "bar"])`). A implementation of Table API can either view rows as a subtype of tables, overload those operators, or give different names to row variants of the operators.

Assumptions

Functions

- even: consumes an integer and returns a boolean
- length: consumes a sequence and measures its length
- schema: extracts the schema of a table
- range: consumes a number and produces a sequence of valid indices
- concat: concatenates two sequences or two strings
- startsWith: checks whether a string starts with another string
- average: computes the average of a sequence of numbers
- filter: the conventional sequence (e.g. lists) filter
- map: the conventional sequence (e.g. lists) map
- removeDuplicates: consumes a sequence and produces a subsequence with all duplicated elements removed
- removeAll: consumes two sequences and produces a subsequence of the first input, removing all elements that also appear in the second input.
- colNameOfNumber: converts a Number to a ColName

Relations

- x has no duplicates
- x is equal to y
- x is (not) in y
- x is a subsequence of y
- x is of sort y
- x is y
- x is a categorical sort
- x is (non-)negative
- x is equal to the sort of y
- x is the sort of elements of y
- x is equal to y with all a_i replaced with b_i

B.3.1 Constructors

emptyTable :: t:Table

Constraints

Requires:

Ensures:

- schema(t) is equal to []
- nrows(t) is equal to 0

Description Create an empty table.

addRows :: t1:Table * rs:Seq<Row> -> t2:Table

Constraints

Requires:

- for all r in rs, schema(r) is equal to schema(t1)

Ensures:

- schema(t2) is equal to schema(t1)
- nrows(t2) is equal to nrows(t1) + length(rs)

Description Consumes a Table and a sequence of Row to add, and produces a new Table with the rows from the original table followed by the given Rows.

```
> addRows(
    students,
    [
      [row:
        ("name", "Colton"), ("age", 19),
        ("favorite color", "blue")]
    ])
| name     | age | favorite color |
| -------- | --- | -------------- |
| "Bob"    | 12  | "blue"         |
| "Alice"  | 17  | "green"        |
| "Eve"    | 13  | "red"          |
| "Colton" | 19  | "blue"         |
> addRows(gradebook, [])
| name    | age | quiz1 | quiz2 | midterm | quiz3 | quiz4 | final |
| ------- | --- | ----- | ----- | ------- | ----- | ----- | ----- |
| "Bob"   | 12  | 8     | 9     | 77      | 7     | 9     | 87    |
| "Alice" | 17  | 6     | 8     | 88      | 8     | 7     | 85    |
| "Eve"   | 13  | 7     | 9     | 84      | 8     | 8     | 77    |
```

addColumn :: t1:Table * c:ColName * vs:Seq<Value> -> t2:Table

Constraints
Requires:

- c is not in header(t1)
- length(vs) is equal to nrows(t1)

Ensures:

- header(t2) is equal to concat(header(t1), [c])
- for all c' in header(t1), schema(t2)[c'] is equal to schema(t1)[c']
- schema(t2)[c] is the sort of elements of vs
- nrows(t2) is equal to nrows(t1)

Description Consumes a column name and a Seq of values and produces a new Table with the columns of the input Table followed by a column with the given name and values. Note that the length of vs must equal the length of the Table.

```
> hairColor = ["brown", "red", "blonde"]
> addColumn(students, "hair-color", hairColor)
```

```
| name    | age | favorite color | hair-color |
| ------- | --- | -------------- | ---------- |
| "Bob"   | 12  | "blue"         | "brown"    |
| "Alice" | 17  | "green"        | "red"      |
| "Eve"   | 13  | "red"          | "blonde"   |
> presentation = [9, 9, 6]
> addColumn(gradebook, "presentation", presentation)
| name    | age | quiz1 | quiz2 | midterm | quiz3 | quiz4 | final | presentation |
| ------- | --- | ----- | ----- | ------- | ----- | ----- | ----- | ------------ |
| "Bob"   | 12  | 8     | 9     | 77      | 7     | 9     | 87    | 9            |
| "Alice" | 17  | 6     | 8     | 88      | 8     | 7     | 85    | 9            |
| "Eve"   | 13  | 7     | 9     | 84      | 8     | 8     | 77    | 6            |
```

buildColumn :: t1:Table * c:ColName * f:(r:Row -> v:Value) -> t2:Table

Constraints
Requires:

- c is not in header(t1)

Ensures:

- schema(r) is equal to schema(t1)
- header(t2) is equal to concat(header(t1), [c])
- for all c' in header(t1), schema(t2)[c'] is equal to schema(t1)[c']
- schema(t2)[c] is equal to the sort of v
- nrows(t2) is equal to nrows(t1)

Description Consumes an existing Table and produces a new Table containing an additional column with the given ColName, using f to compute the values for that column, once for each row.

```
> isTeenagerBuilder =
    function(r):
      12 < getValue(r, "age") and getValue(r, "age") < 20
    end
> buildColumn(students, "is-teenager", isTeenagerBuilder)
| name    | age | favorite color | is-teenager |
| ------- | --- | -------------- | ----------- |
| "Bob"   | 12  | "blue"         | false       |
| "Alice" | 17  | "green"        | true        |
| "Eve"   | 13  | "red"          | true        |
> didWellInFinal =
    function(r):
      85 <= getValue(r, "final")
    end
> buildColumn(gradebook, "did-well-in-final", didWellInFinal)
| name    | age | quiz1 | quiz2 | midterm | quiz3 | quiz4 | final | did-well-in-final |
| ------- | --- | ----- | ----- | ------- | ----- | ----- | ----- | ----------------- |
| "Bob"   | 12  | 8     | 9     | 77      | 7     | 9     | 87    | true              |
| "Alice" | 17  | 6     | 8     | 88      | 8     | 7     | 85    | true              |
| "Eve"   | 13  | 7     | 9     | 84      | 8     | 8     | 77    | false             |
```

vcat :: t1:Table * t2:Table -> t3:Table

Constraints
Requires:

- schema(t1) is equal to schema(t2)

Ensures:

- schema(t3) is equal to schema(t1)
- nrows(t3) is equal to nrows(t1) + nrows(t2)

Description Combines two tables vertically. The output table starts with rows from the first input table, followed by the rows from the second input table.

```
> increaseAge =
    function(r):
      [row: ("age", 1 + getValue(r, "age"))]
    end
> vcat(students, update(students, increaseAge))
| name    | age | favorite color |
| ------- | --- | -------------- |
| "Bob"   | 12  | "blue"         |
| "Alice" | 17  | "green"        |
| "Eve"   | 13  | "red"          |
| "Bob"   | 13  | "blue"         |
| "Alice" | 18  | "green"        |
| "Eve"   | 14  | "red"          |
> curveMidtermAndFinal =
    function(r):
      curve =
        function(n):
          n + 5
        end
      [row:
        ("midterm", curve(getValue("midterm"))),
        ("final", curve(getValue("final")))]
    end
> vcat(gradebook, update(gradebook, curveMidtermAndFinal))
| name    | age | quiz1 | quiz2 | midterm | quiz3 | quiz4 | final |
| ------- | --- | ----- | ----- | ------- | ----- | ----- | ----- |
| "Bob"   | 12  | 8     | 9     | 77      | 7     | 9     | 87    |
| "Alice" | 17  | 6     | 8     | 88      | 8     | 7     | 85    |
| "Eve"   | 13  | 7     | 9     | 84      | 8     | 8     | 77    |
| "Bob"   | 12  | 8     | 9     | 82      | 7     | 9     | 92    |
| "Alice" | 17  | 6     | 8     | 93      | 8     | 7     | 90    |
| "Eve"   | 13  | 7     | 9     | 89      | 8     | 8     | 82    |
```

hcat :: t1:Table * t2:Table -> t3:Table

Constraints
Requires:

- concat(header(t1), header(t2)) has no duplicates
- nrows(t1) is equal to nrows(t2)

Ensures:

- schema(t3) is equal to concat(schema(t1), schema(t2))
- nrows(t3) is equal to nrows(t1)

Description Combines two tables horizontally. The output table starts with columns from the first input, followed by the columns from the second input.

```
> hcat(students, dropColumns(gradebook, ["name", "age"]))
| name    | age | favorite color | quiz1 | quiz2 | midterm | quiz3 | quiz4 | final |
| ------- | --- | -------------- | ----- | ----- | ------- | ----- | ----- | ----- |
| "Bob"   | 12  | "blue"         | 8     | 9     | 77      | 7     | 9     | 87    |
| "Alice" | 17  | "green"        | 6     | 8     | 88      | 8     | 7     | 85    |
| "Eve"   | 13  | "red"          | 7     | 9     | 84      | 8     | 8     | 77    |
> hcat(dropColumns(students, ["name", "age"]), gradebook)
| favorite color | name    | age | quiz1 | quiz2 | midterm | quiz3 | quiz4 | final |
| -------------- | ------- | --- | ----- | ----- | ------- | ----- | ----- | ----- |
| "blue"         | "Bob"   | 12  | 8     | 9     | 77      | 7     | 9     | 87    |
| "green"        | "Alice" | 17  | 6     | 8     | 88      | 8     | 7     | 85    |
| "red"          | "Eve"   | 13  | 7     | 9     | 84      | 8     | 8     | 77    |
```

values :: rs:Seq<Row> -> t:Table

Constraints
Requires:

- length(rs) is positive
- for all r in rs, schema(r) is equal to schema(rs[0])

Ensures:

- schema(t) is equal to schema(rs[0])
- nrows(t) is equal to length(rs)

Description Returns a sequence of one or more rows as a table.

```
> values([
    [row: ("name", "Alice")],
    [row: ("name", "Bob")]])
| name    |
| ------- |
| "Alice" |
| "Bob"   |
> values([
    [row: ("name", "Alice"), ("age", 12)],
    [row: ("name", "Bob"), ("age", 13)]])
| name    | age |
| ------- | --- |
| "Alice" | 12  |
| "Bob"   | 13  |
```

crossJoin :: t1:Table * t2:Table -> t3:Table

Constraints
Requires:

- concat(header(t1), header(t2)) has no duplicates

Ensures:

- schema(t3) is equal to concat(schema(t1), schema(t2))
- nrows(t3) is equal to nrows(t1) * nrows(t2)

Description Computes the cartesian product of two tables.

```
> petiteJelly = subTable(jellyAnon, [0, 1], [0, 1, 2])
> petiteJelly
| get acne | red   | black |
| -------- | ----- | ----- |
| true     | false | false |
| true     | false | true  |
> crossJoin(students, petiteJelly)
| name    | age | favorite color | get acne | red   | black |
| ------- | --- | -------------- | -------- | ----- | ----- |
| "Bob"   | 12  | "blue"         | true     | false | false |
| "Bob"   | 12  | "blue"         | true     | false | true  |
| "Alice" | 17  | "green"        | true     | false | false |
| "Alice" | 17  | "green"        | true     | false | true  |
| "Eve"   | 13  | "red"          | true     | false | false |
| "Eve"   | 13  | "red"          | true     | false | true  |
> crossJoin(emptyTable, petiteJelly)
| get acne | red   | black |
| -------- | ----- | ----- |
```

leftJoin :: t1:Table * t2:Table * cs:Seq<ColName> -> t3:Table

Constraints
Requires:

- cs has no duplicates
- for all c in cs, c is in header(t1)
- for all c in cs, c is in header(t2)
- for all c in cs, schema(t1)[c] is equal to schema(t2)[c]
- concat(header(t1), removeAll(header(t2), cs)) has no duplicates

Ensures:

- header(t3) is equal to concat(header(t1), removeAll(header(t2), cs))
- for all c in header(t1), schema(t3)[c] is equal to schema(t1)[c]
- for all c in removeAll(header(t2), cs)), schema(t3)[c] is equal to schema(t2)[c]
- nrows(t3) is equal to nrows(t1)

Description Looks up more information on rows of the first table and add those information to create a new table. The named columns define the keys for looking up. If there is no corresponding row in t2, the extra column will be filled with empty cells.

```
> leftJoin(students, gradebook, ["name", "age"])
| name    | age | favorite color | quiz1 | quiz2 | midterm | quiz3 | quiz4 | final |
| ------- | --- | -------------- | ----- | ----- | ------- | ----- | ----- | ----- |
| "Bob"   | 12  | "blue"         | 8     | 9     | 77       | 7     | 9     | 87     |
| "Alice" | 17  | "green"        | 6     | 8     | 88       | 8     | 7     | 85     |
| "Eve"   | 13  | "red"          | 7     | 9     | 84       | 8     | 8     | 77     |
> leftJoin(employees, departments, ["Department ID"])
| Last Name    | Department ID | Department Name |
| ------------ | ------------- | --------------- |
| "Rafferty"   | 31            | "Sales"         |
| "Jones"      | 32            |                 |
| "Heisenberg" | 33            | "Engineering"   |
| "Robinson"   | 34            | "Clerical"      |
| "Smith"      | 34            | "Clerical"      |
| "Williams"   |               |                 |
```

B.3.2 Properties

nrows :: t:Table -> n:Number

Constraints
Requires:

Ensures:

- n is equal to nrows(t)

Description Returns a Number representing the number of rows in the Table.

```
> nrows(emptyTable)
0
> nrows(studentsMissing)
3
```

ncols :: t:Table -> n:Number

Constraints
Requires:

Ensures:

- n is equal to ncols(t)

Description Returns a Number representing the number of columns in the Table.

```
> ncols(students)
3
> ncols(studentsMissing)
3
```

header :: t:Table -> cs:Seq<ColName>

Constraints
Requires:

Ensures:

- cs is equal to header(t)

Description Returns a Seq representing the column names in the Table.

```
> header(students)
["name", "age", "favorite color"]
> header(gradebook)
["name", "age", "quiz1", "quiz2", "midterm", "quiz3", "quiz4", "final"]
```

B.3.3 Access Subcomponents

getRow :: t:Table * n:Number -> r:Row

Constraints
Requires:

- n is in range(nrows(t))

Ensures:

Description Extracts a row out of a table by a numeric index.

```
> getRow(students, 0)
[row: ("name", "Bob"), ("age", 12), ("favorite color", "blue")]
> getRow(gradebook, 1)
[row:
  ("name", "Alice"), ("age", 17),
  ("quiz1", 6), ("quiz2", 8), ("midterm", 88),
  ("quiz3", 8), ("quiz4", 7), ("final", 85)]
```

getValue :: r:Row * c:ColName -> v:Value

Constraints
Requires:

- c is in header(r)

Ensures:

- v is of sort schema(r)[c]

Description Retrieves the value for the column c in the row r.

```
> getValue([row: ("name", "Bob"),  ("age", 12)], "name")
"Bob"
> getValue([row: ("name", "Bob"),  ("age", 12)], "age")
12
```

(overloading 1/2) getColumn :: t:Table * n:Number -> vs:Seq<Value>

Constraints
Requires:

- n is in range(ncols(t))

Ensures:

- length(vs) is equal to nrows(t)
- for all v in vs, v is of sort schema(t)[header(t)[n]]

Description Returns a Seq of the values in the indexed column in t.

```
> getColumn(students, 1)
[12, 17, 13]
> getColumn(gradebook, 0)
["Bob", "Alice", "Eve"]
```

(overloading 2/2) getColumn :: t:Table * c:ColName -> vs:Seq<Value>

Constraints
Requires:

- c is in header(t)

Ensures:

- for all v in vs, v is of sort schema(t)[c]
- length(vs) is equal to nrows(t)

Description Returns a Seq of the values in the named column in t.

```
> getColumn(students, "age")
[12, 17, 13]
> getColumn(gradebook, "name")
["Bob", "Alice", "Eve"]
```

B.3.4 Subtable

(overload 1/2) selectRows :: t1:Table * ns:Seq<Number> -> t2:Table

Constraints
Requires:

- for all n in ns, n is in range(nrows(t1))

Ensures:

- schema(t2) is equal to schema(t1)
- nrows(t2) is equal to length(ns)

Description Given a Table and a Seq<Number> containing row indices, produces a new Table containing only those rows.

```
> selectRows(students, [2, 0, 2, 1])
| name    | age | favorite color |
| ------- | --- | -------------- |
| "Eve"   | 13  | "red"          |
| "Bob"   | 12  | "blue"         |
| "Eve"   | 13  | "red"          |
| "Alice" | 17  | "green"        |
> selectRows(gradebooks, [2, 1])
| name    | age | quiz1 | quiz2 | midterm | quiz3 | quiz4 | final |
| ------- | --- | ----- | ----- | ------- | ----- | ----- | ----- |
| "Eve"   | 13  | 7     | 9     | 84      | 8     | 8     | 77    |
| "Alice" | 17  | 6     | 8     | 88      | 8     | 7     | 85    |
```

(overload 2/2) selectRows :: t1:Table * bs:Seq<Boolean> -> t2:Table

Constraints
Requires:

- length(bs) is equal to nrows(t1)

Ensures:

- schema(t2) is equal to schema(t1)
- nrows(t2) is equal to length(removeAll(bs, [false]))

Description Given a Table and a Seq<Boolean> that represents a predicate on rows, returns a Table with only the rows for which the predicate returns true.

```
> selectRows(students, [true, false, true])
| name  | age | favorite color |
| ----- | --- | -------------- |
| "Bob" | 12  | "blue"         |
| "Eve" | 13  | "red"          |
> selectRows(gradebook, [false, false, true])
| name  | age | quiz1 | quiz2 | midterm | quiz3 | quiz4 | final |
| ----- | --- | ----- | ----- | ------- | ----- | ----- | ----- |
| "Eve" | 13  | 7     | 9     | 84      | 8     | 8     | 77    |
```

(overload 1/3) selectColumns :: t1:Table * bs:Seq<Boolean> -> t2:Table

Constraints
Requires:

- length(bs) is equal to ncols(t1)

Ensures:

- header(t2) is a subsequence of header(t1)
- for all i in range(ncols(t1)), header(t1)[i] is in header(t2) if and only if bs[i] is equal to true

- schema(t2) is a subsequence of schema(t1)
- nrows(t2) is equal to nrows(t1)

Description Consumes a Table and a Seq<Boolean> deciding whether each column should be kept, and produces a new Table containing only those columns. The order of the columns is as in the input table.

```
> selectColumns(students, [true, true, false])
| name    | age |
| ------- | --- |
| "Bob"   | 12  |
| "Alice" | 17  |
| "Eve"   | 13  |
> selectColumns(gradebook, [true, false, false, false, true, false, false, true])
| name    | midterm | final |
| ------- | ------- | ----- |
| "Bob"   | 77      | 87    |
| "Alice" | 88      | 85    |
| "Eve"   | 84      | 77    |
```

(overload 2/3) selectColumns :: t1:Table * ns:Seq<Number> -> t2:Table

Constraints
Requires:

- ns has no duplicates
- for all n in ns, n is in range(ncols(t1))

Ensures:

- ncols(t2) is equal to length(ns)
- for all i in range(length(ns)), header(t2)[i] is equal to header(t1)[ns[i]]
- for all c in header(t2), schema(t2)[c] is equal to schema(t1)[c]
- nrows(t2) is equal to nrows(t1)

Description Consumes a Table and a Seq<Number> containing column indices, and produces a new Table containing only those columns. The order of the columns is as given in the input Seq.

```
> selectColumns(students, [2, 1])
| favorite color | age |
| -------------- | --- |
| "blue"         | 12  |
| "green"        | 17  |
| "red"          | 13  |
> selectColumns(gradebook, [7, 0, 4])
| final | name    | midterm |
| ----- | ------- | ------- |
| 87    | "Bob"   | 77      |
| 85    | "Alice" | 88      |
| 77    | "Eve"   | 84      |
```

(overload 3/3) selectColumns :: t1:Table * cs:Seq<ColName> -> t2:Table

Constraints
Requires:

- cs has no duplicates
- for all c in cs, c is in header(t1)

Ensures:

- header(t2) is equal to cs
- for all c in header(t2), schema(t2)[c] is equal to schema(t1)[c]
- nrows(t2) is equal to nrows(t1)

Description Consumes a Table and a Seq<ColName> containing column names, and produces a new Table containing only those columns. The order of the columns is as given in the input Seq.

```
> selectColumns(students, ["favorite color", "age"])
| favorite color | age |
| -------------- | --- |
| "blue"         | 12  |
| "green"        | 17  |
| "red"          | 13  |
> selectColumns(gradebook, ["final", "name", "midterm"])
| final | name    | midterm |
| ----- | ------- | ------- |
| 87    | "Bob"   | 77      |
| 85    | "Alice" | 88      |
| 77    | "Eve"   | 84      |
```

head :: t1:Table * n:Number -> t2:Table

Constraints
Requires:

- if n is non-negative then n is in range(nrows(t1))
- if n is negative then - n is in range(nrows(t1))

Ensures:

- schema(t2) is equal to schema(t1)
- if n is non-negative then nrows(t2) is equal to n
- if n is negative then nrows(t2) is equal to nrows(t1) + n

Description Returns the first n rows of the table based on position. For negative values of n, this function returns all rows except the last n rows.

```
> head(students, 1)
| name    | age | favorite color |
| ------- | --- | -------------- |
| "Bob"   | 12  | "blue"         |
> head(students, -2)
| name    | age | favorite color |
| ------- | --- | -------------- |
| "Bob"   | 12  | "blue"         |
```

distinct :: t1:Table -> t2:Table

Constraints
Requires:

Ensures:

- schema(t2) is equal to schema(t1)

Description Retains only unique/distinct rows from an input Table.

```
> distinct(students)
| name    | age | favorite color |
| ------- | --- | -------------- |
| "Bob"   | 12  | "blue"         |
| "Alice" | 17  | "green"        |
| "Eve"   | 13  | "red"          |
> distinct(selectColumns(gradebook, ["quiz3"]))
| quiz3 |
| ----- |
| 7     |
| 8     |
```

dropColumn :: t1:Table * c:ColName -> t2:Table

Constraints
Requires:

- c is in header(t1)

Ensures:

- nrows(t2) is equal to nrows(t1)
- header(t2) is equal to removeAll(header(t1), [c])
- schema(t2) is a subsequence of schema(t1)

Description Returns a Table that is the same as t, except without the named column.

```
> dropColumn(students, "age")
| name    | favorite color |
| ------- | -------------- |
| "Bob"   | "blue"         |
| "Alice" | "green"        |
| "Eve"   | "red"          |
> dropColumn(gradebook, "final")
| name    | age | quiz1 | quiz2 | midterm | quiz3 | quiz4 |
| ------- | --- | ----- | ----- | ------- | ----- | ----- |
| "Bob"   | 12  | 8     | 9     | 77      | 7     | 9     |
| "Alice" | 17  | 6     | 8     | 88      | 8     | 7     |
| "Eve"   | 13  | 7     | 9     | 84      | 8     | 8     |
```

dropColumns :: t1:Table * cs:Seq<ColName> -> t2:Table

Constraints
Requires:

- for all c in cs, c is in header(t1)
- cs has no duplicates

Ensures:

- nrows(t2) is equal to nrows(t1)
- header(t2) is equal to removeAll(header(t1), cs)
- schema(t2) is a subsequence of schema(t1)

Description Returns a Table that is the same as t, except without the named columns.

```
> dropColumns(students, ["age"])
| name    | favorite color |
| ------- | -------------- |
| "Bob"   | "blue"         |
| "Alice" | "green"        |
| "Eve"   | "red"          |
> dropColumns(gradebook, ["final", "midterm"])
| name    | age | quiz1 | quiz2 | quiz3 | quiz4 |
| ------- | --- | ----- | ----- | ----- | ----- |
| "Bob"   | 12  | 8     | 9     | 7     | 9     |
| "Alice" | 17  | 6     | 8     | 8     | 7     |
| "Eve"   | 13  | 7     | 9     | 8     | 8     |
```

tfilter :: t1:Table * f:(r:Row -> b:Boolean) -> t2:Table

Constraints
Requires:

Ensures:

- schema(r) is equal to schema(t1)
- schema(t2) is equal to schema(t1)

Description Given a Table and a predicate on rows, returns a Table with only the rows for which the predicate returns true.

```
> ageUnderFifteen =
    function(r):
      getValue(r, "age") < 15
    end
> tfilter(students, ageUnderFifteen)
| name  | age | favorite color |
| ----- | --- | -------------- |
| "Bob" | 12  | "blue"         |
| "Eve" | 13  | "red"          |
> nameLongerThan3Letters =
    function(r):
      length(getValue(r, "name")) > 3
    end
> tfilter(gradebook, nameLongerThan3Letters)
| name    | age | quiz1 | quiz2 | midterm | quiz3 | quiz4 | final |
| ------- | --- | ----- | ----- | ------- | ----- | ----- | ----- |
| "Alice" | 17  | 6     | 8     | 88      | 8     | 7     | 85    |
```

B.3.5 Ordering

tsort :: t1:Table * c:ColName * b:Boolean -> t2:Table

Constraints
Requires:

- c is in header(t1)
- schema(t1)[c] is Number

Ensures:

- nrows(t2) is equal to nrows(t1)
- schema(t2) is equal to schema(t1)

Description Given a Table and one of its column names, returns a Table with the same rows ordered based on the named column. If b is true, the Table will be sorted in ascending order, otherwise it will be in descending order.

```
> tsort(students, "age", true)
| name    | age | favorite color |
| ------- | --- | -------------- |
| "Bob"   | 12  | "blue"         |
| "Eve"   | 13  | "red"          |
| "Alice" | 17  | "green"        |
> tsort(gradebook, "final", false)
| name    | age | quiz1 | quiz2 | midterm | quiz3 | quiz4 | final |
| ------- | --- | ----- | ----- | ------- | ----- | ----- | ----- |
| "Bob"   | 12  | 8     | 9     | 77      | 7     | 9     | 87    |
| "Alice" | 17  | 6     | 8     | 88      | 8     | 7     | 85    |
| "Eve"   | 13  | 7     | 9     | 84      | 8     | 8     | 77    |
```

sortByColumns :: t1:Table * cs:Seq<ColName> -> t2:Table

Constraints
Requires:

- cs has no duplicates
- for all c in cs, c is in header(t1)
- for all c in cs, schema(t1)[c] is Number

Ensures:

- nrows(t2) is equal to nrows(t1)
- schema(t2) is equal to schema(t1)

Description Given a Table and a sequence of column names in that Table, return a Table with the same rows ordered ascendingly based on the named columns.

```
> sortByColumns(students, ["age"])
| name    | age | favorite color |
| ------- | --- | -------------- |
| "Bob"   | 12  | "blue"         |
| "Eve"   | 13  | "red"          |
| "Alice" | 17  | "green"        |
> sortByColumns(gradebook, ["quiz2", "quiz1"])
| name    | age | quiz1 | quiz2 | midterm | quiz3 | quiz4 | final |
| ------- | --- | ----- | ----- | ------- | ----- | ----- | ----- |
| "Alice" | 17  | 6     | 8     | 88      | 8     | 7     | 85    |
| "Eve"   | 13  | 7     | 9     | 84      | 8     | 8     | 77    |
| "Bob"   | 12  | 8     | 9     | 77      | 7     | 9     | 87    |
```

orderBy :: t1:Table * Seq<Exists K . getKey:(r:Row -> k:K) * compare:(k1:K * k2:K -> Boolean)> -> t2:Table

Constraints
Requires:

Ensures:

- schema(r) is equal to schema(t1)
- schema(t2) is equal to schema(t1)
- nrows(t2) is equal to nrows(t1)

Description Sorts the rows of a Table in ascending order by using a sequence of specified comparers.

```
> nameLength =
    function(r):
      length(getValue(r, "name"))
    end
> le =
    function(n1, n2):
      n1 <= n2
    end
> orderBy(students, [(nameLength, le)])
| name    | age | favorite color |
| ------- | --- | -------------- |
| "Bob"   | 12  | "blue"         |
| "Eve"   | 13  | "red"          |
| "Alice" | 17  | "green"        |
> midtermAndFinal =
    function(r):
      [getValue(r, "midterm"), getValue(r, "final")]
    end
> compareGrade =
    function(g1, g2):
      le(average(g1), average(g2))
    end
> orderBy(gradebook, [(nameLength, ge), (midtermAndFinal, compareGrade)])
| name    | age | quiz1 | quiz2 | midterm | quiz3 | quiz4 | final |
| ------- | --- | ----- | ----- | ------- | ----- | ----- | ----- |
| "Alice" | 17  | 6     | 8     | 88      | 8     | 7     | 85    |
| "Eve"   | 13  | 7     | 9     | 84      | 8     | 8     | 77    |
| "Bob"   | 12  | 8     | 9     | 77      | 7     | 9     | 87    |
```

B.3.6 Aggregate

count :: t1:Table * c:ColName -> t2:Table

Constraints
Requires:

- c is in header(t1)
- schema(t1)[c] is a categorical sort

Ensures:

- header(t2) is equal to ["value", "count"]
- schema(t2)["value"] is equal to schema(t1)[c]
- schema(t2)["count"] is equal to Number
- nrows(t2) is equal to length(removeDuplicates(getColumn(t1, c)))

Description Takes a Table and a ColName representing the name of a column in that Table. Produces a Table that summarizes how many rows have each value in the given column.

```
> count(students, "favorite color")
| value   | count |
| ------- | ----- |
| "blue"  | 1     |
| "green" | 1     |
| "red"   | 1     |
> count(gradebook, "age")
| value | count |
| ----- | ----- |
| 12    | 1     |
| 17    | 1     |
| 13    | 1     |
```

bin :: t1:Table * c:ColName * n:Number -> t2:Table

Constraints
Requires:

- c is in header(t1)
- schema(t1)[c] is Number

Ensures:

- header(t2) is equal to ["group", "count"]
- schema(t2)["group"] is String
- schema(t2)["count"] is Number

Description Groups the values of a numeric column into bins. The parameter n specifies the bin width. This function is useful in creating histograms and converting continuous random variables to categorical ones.

```
> bin(students, "age", 5)
| group            | count |
| ---------------- | ----- |
| "10 <= age < 15" | 2     |
| "15 <= age < 20" | 1     |
> bin(gradebook, "final", 5)
| group            | count |
| ---------------- | ----- |
| "75 <= age < 80" | 1     |
| "80 <= age < 85" | 0     |
| "85 <= age < 90" | 2     |
```

pivotTable :: t1:Table * cs:Seq<ColName> * aggs:Seq<ColName * ColName * Function> -> t2:Table

Constraints
Let ci1 and ci2 and fi be the components of aggs[i] for all i in range(length(aggs))
Requires:

- for all c in cs, c is in header(t1)
- for all c in cs, schema(t1)[c] is a categorical sort
- ci2 is in header(t1)
- concat(cs, [c11, ... , cn1]) has no duplicates

Ensures:

- fi consumes Seq<schema(t1)[ci2]>
- header(t2) is equal to concat(cs, [c11, ... , cn1])
- for all c in cs, schema(t2)[c] is equal to schema(t1)[c]
- schema(t2)[ci1] is equal to the sort of outputs of fi for all i

Description Partitions rows into groups and summarize each group with the functions in agg. Each element of agg specifies the output column, the input column, and the function that compute the summarizing value (e.g. average, sum, and count).

```
> pivotTable(students, ["favorite color"], [("age-average", "age", average)])
| favorite color | age-average |
| -------------- | ----------- |
| "blue"         | 12          |
| "green"        | 17          |
| "red"          | 13          |
> proportion =
    function(bs):
      n = length(filter(bs, function(b): b end))
      n / length(bs)
    end
> pivotTable(
    jellyNamed,
    ["get acne", "brown"],
    [
      ("red proportion", "red", proportion),
      ("pink proportion", "pink", proportion)
    ])
| get acne | brown | red proportion | pink proportion |
| -------- | ----- | -------------- | --------------- |
| false    | false | 0              | 3/4             |
| false    | true  | 1              | 1               |
| true     | false | 0              | 1/4             |
| true     | true  | 0              | 0               |
```

groupBy<K,V> :: t1:Table * key:(r1:Row -> k1:K) * project:(r2:Row -> v:V) * aggregate:(k2:K * vs:Seq<V> -> r3:Row) -> t2:Table

Constraints
Requires:

Ensures:

- schema(r1) is equal to schema(t1)
- schema(r2) is equal to schema(t1)
- schema(t2) is equal to schema(r3)
- nrows(t2) is equal to length(removeDuplicates(ks)), where ks is the results of applying key to each row of t1. ks can be defined with select and getColumn.

Description Groups the rows of a table according to a specified key selector function and creates a result value from each group and its key. The rows of each group are projected by using a specified function.

```
> colorTemp =
    function(r):
      if getValue(r, "favorite color") == "red":
        "warm"
      else:
        "cool"
      end
    end
> nameLength =
    function(r):
      length(getValue(r, "name"))
    end
> aggregate =
    function(k, vs):
      [row: ("key", k), ("average", average(vs))]
    end
> groupBy(students, colorTemp, nameLength, aggregate)
| key    | average |
| ------ | ------- |
| "warm" | 3       |
| "cool" | 4       |
> abstractAge =
    function(r):
      if (getValue(r, "age") <= 12):
        "kid"
      else if (getValue(r, "age") <= 19):
        "teenager"
      else:
        "adult"
      end
    end
> finalGrade =
    function(r):
      getValue(r, "final")
    end
> groupBy(gradebook, abstractAge, finalGrade, aggregate)
| key        | average |
| ---------- | ------- |
| "kid"      | 87      |
| "teenager" | 81      |
```

B.3.7 Missing values

completeCases :: t:Table * c:ColName -> bs:Seq<Boolean>

Constraints
Requires:

- c is in header(t)

Ensures:

- length(bs) is equal to nrows(t)

Description Return a Seq<Boolean> with true entries indicating rows without missing values (complete cases) in table t.

```
> completeCases(students, "age")
[true, true, true]
> completeCases(studentsMissing, "age")
[false, true, true]
```

dropna :: t1:Table -> t2:Table

Constraints
Requires:

Ensures:

- schema(t2) is equal to schema(t1)

Description Removes rows that have some values missing

```
> dropna(studentsMissing)
| name    | age | favorite color |
| ------- | --- | -------------- |
| "Alice" | 17  | "green"        |
> dropna(gradebookMissing)
| name  | age | quiz1 | quiz2 | midterm | quiz3 | quiz4 | final |
| ----- | --- | ----- | ----- | ------- | ----- | ----- | ----- |
| "Bob" | 12  | 8     | 9     | 77      | 7     | 9     | 87    |
```

fillna :: t1:Table * c:ColName * v:Value -> t2:Table

Constraints
Requires:

- c is in header(t1)
- v is of sort schema(t1)[c]

Ensures:

- schema(t2) is equal to schema(t1)
- nrows(t2) is equal to nrows(t1)

Description Scans the named column and fills in v when a cell is missing value.

```
> fillna(studentsMissing, "favorite color", "white")
| name    | age | favorite color |
| ------- | --- | -------------- |
| "Bob"   |     | "blue"         |
| "Alice" | 17  | "green"        |
| "Eve"   | 13  | "white"        |
> fillna(gradebookMissing, "quiz1", 0)
| name    | age | quiz1 | quiz2 | midterm | quiz3 | quiz4 | final |
| ------- | --- | ----- | ----- | ------- | ----- | ----- | ----- |
| "Bob"   | 12  | 8     | 9     | 77      | 7     | 9     | 87    |
| "Alice" | 17  | 6     | 8     | 88      |       | 7     | 85    |
| "Eve"   | 13  | 0     | 9     | 84      | 8     | 8     | 77    |
```

B.3.8 Data Cleaning

pivotLonger :: t1:Table * cs:Seq<ColName> * c1:ColName * c2:ColName -> t2:Table

Constraints
Requires:

- length(cs) is positive
- cs has no duplicates
- for all c in cs, c is in header(t1)
- for all c in cs, schema(t1)[c] is equal to schema(t1)[cs[0]]
- concat(removeAll(header(t1), cs), [c1, c2]) has no duplicates

Ensures:

- header(t2) is equal to concat(removeAll(header(t1), cs), [c1, c2])
- for all c in removeAll(header(t1), cs), schema(t2)[c] is equal to schema(t1)[c]
- schema(t2)[c1] is equal to ColName
- schema(t2)[c2] is equal to schema(t1)[cs[0]]

Description Reshapes the input table and make it longer. The data kept in the named columns are moved to two new columns, one for the column names and the other for the cell values.

```
> pivotLonger(gradebook, ["midterm", "final"], "exam", "score")
| name    | age | quiz1 | quiz2 | quiz3 | quiz4 | exam      | score |
| ------- | --- | ----- | ----- | ----- | ----- | --------- | ----- |
| "Bob"   | 12  | 8     | 9     | 7     | 9     | "midterm" | 77    |
| "Bob"   | 12  | 8     | 9     | 7     | 9     | "final"   | 87    |
| "Alice" | 17  | 6     | 8     | 8     | 7     | "midterm" | 88    |
| "Alice" | 17  | 6     | 8     | 8     | 7     | "final"   | 85    |
| "Eve"   | 13  | 7     | 9     | 8     | 8     | "midterm" | 84    |
| "Eve"   | 13  | 7     | 9     | 8     | 8     | "final"   | 77    |
> pivotLonger(gradebook, ["quiz1", "quiz2", "quiz3", "quiz4", "midterm",
  "final"], "test", "score")
| name    | age | test      | score |
| ------- | --- | --------- | ----- |
| "Bob"   | 12  | "quiz1"   | 8     |
| "Bob"   | 12  | "quiz2"   | 9     |
| "Bob"   | 12  | "quiz3"   | 7     |
| "Bob"   | 12  | "quiz4"   | 9     |
| "Bob"   | 12  | "midterm" | 77    |
| "Bob"   | 12  | "final"   | 87    |
| "Alice" | 17  | "quiz1"   | 6     |
| "Alice" | 17  | "quiz2"   | 8     |
| "Alice" | 17  | "quiz3"   | 8     |
| "Alice" | 17  | "quiz4"   | 7     |
| "Alice" | 17  | "midterm" | 88    |
| "Alice" | 17  | "final"   | 85    |
| "Eve"   | 13  | "quiz1"   | 7     |
| "Eve"   | 13  | "quiz2"   | 9     |
| "Eve"   | 13  | "quiz3"   | 8     |
| "Eve"   | 13  | "quiz4"   | 8     |
| "Eve"   | 13  | "midterm" | 84    |
| "Eve"   | 13  | "final"   | 77    |
```

pivotWider :: t1:Table * c1:ColName * c2:ColName -> t2:Table

Constraints
Requires:

- c1 is in header(t1)
- c2 is in header(t1)
- schema(t1)[c1] is ColName
- concat(removeAll(header(t1), [c1, c2]), removeDuplicates(getColumn(t1, c1))) has no duplicates

Ensures:

- header(t2) is equal to concat(removeAll(header(t1), [c1, c2]), removeDuplicates(getColumn(t1, c1)))
- for all c in removeAll(header(t1), [c1, c2]), schema(t2)[c] is equal to schema(t1)[c]
- for all c in removeDuplicates(getColumn(t1, c1)), schema(t2)[c] is equal to schema(t1)[c2]

Description The inverse of pivotLonger.

```
> pivotWider(students, "name", "age")
| favorite color | Bob | Alice | Eve |
| -------------- | --- | ----- | --- |
| "blue"         | 12  |       |     |
| "green"        |     | 17    |     |
| "red"          |     |       | 13  |
> longerTable =
    pivotLonger(
      gradebook,
      ["quiz1", "quiz2", "quiz3", "quiz4", "midterm", "final"],
      "test",
      "score")
> pivotWider(longerTable, "test", "score")
| name    | age | quiz1 | quiz2 | quiz3 | quiz4 | midterm | final |
| ------- | --- | ----- | ----- | ----- | ----- | ------- | ----- |
| "Bob"   | 12  | 8     | 9     | 7     | 9     | 77      | 87    |
| "Alice" | 17  | 6     | 8     | 8     | 7     | 88      | 85    |
| "Eve"   | 13  | 7     | 9     | 8     | 8     | 84      | 77    |
```

B.3.9 Utilities

flatten :: t1:Table * cs:Seq<ColName> -> t2:Table

Constraints
Requires:

- cs has no duplicates
- for all c in cs, c is in header(t1)
- for all c in cs, schema(t1)[c] is Seq<X> for some sort X
- for all i in range(nrows(t1)), for all c1 and c2 in cs, length(getValue(getRow(t1, i), c1)) is equal to length(getValue(getRow(t1, i), c2))

Ensures:

- header(t2) is equal to header(t1)
- for all c in header(t2)
  - if c is in cs then schema(t2)[c] is equal to the element sort of schema(t1)[c]
  - otherwise, schema(t2)[c] is equal to schema(t1)[c]

Description When columns cs of table t have sequences, returns a Table where each element of each c in cs is flattened, meaning the column corresponding to c becomes a longer column where the original entries are concatenated. Elements of row i of t in columns other than cs will be repeated according to the length of getValue(getRow(t1, i), c1). These lengths must therefore be the same for each c in cs.

```
> flatten(gradebookSeq, ["quizzes"])
| name    | age | quizzes | midterm | final |
| ------- | --- | ------- | ------- | ----- |
| "Bob"   | 12  | 8       | 77      | 87    |
| "Bob"   | 12  | 9       | 77      | 87    |
| "Bob"   | 12  | 7       | 77      | 87    |
| "Bob"   | 12  | 9       | 77      | 87    |
| "Alice" | 17  | 6       | 88      | 85    |
| "Alice" | 17  | 8       | 88      | 85    |
| "Alice" | 17  | 8       | 88      | 85    |
| "Alice" | 17  | 7       | 88      | 85    |
| "Eve"   | 13  | 7       | 84      | 77    |
| "Eve"   | 13  | 9       | 84      | 77    |
| "Eve"   | 13  | 8       | 84      | 77    |
| "Eve"   | 13  | 8       | 84      | 77    |
> t = buildColumn(gradebookSeq, "quiz-pass?",
    function(r):
      isPass =
        function(n):
          n >= 8
        end
      map(getValue(r, "quizzes"), isPass)
    end)
> t
| name    | age | quizzes      | midterm | final | quiz-pass?                 |
| ------- | --- | ------------ | ------- | ----- | -------------------------- |
| "Bob"   | 12  | [8, 9, 7, 9] | 77      | 87    | [true, true, false, true]  |
| "Alice" | 17  | [6, 8, 8, 7] | 88      | 85    | [false, true, true, false] |
| "Eve"   | 13  | [7, 9, 8, 8] | 84      | 77    | [false, true, true, true]  |
> flatten(t, ["quiz-pass?", "quizzes"])
| name    | age | quizzes | midterm | final | quiz-pass? |
| ------- | --- | ------- | ------- | ----- | ---------- |
| "Bob"   | 12  | 8       | 77      | 87    | true       |
| "Bob"   | 12  | 9       | 77      | 87    | true       |
| "Bob"   | 12  | 7       | 77      | 87    | false      |
| "Bob"   | 12  | 9       | 77      | 87    | true       |
| "Alice" | 17  | 6       | 88      | 85    | false      |
| "Alice" | 17  | 8       | 88      | 85    | true       |
| "Alice" | 17  | 8       | 88      | 85    | true       |
| "Alice" | 17  | 7       | 88      | 85    | false      |
| "Eve"   | 13  | 7       | 84      | 77    | false      |
| "Eve"   | 13  | 9       | 84      | 77    | true       |
| "Eve"   | 13  | 8       | 84      | 77    | true       |
| "Eve"   | 13  | 8       | 84      | 77    | true       |
```

transformColumn :: t1:Table * c:ColName * f:(v1:Value -> v2:Value) -> t2:Table

Constraints

Requires:

- c is in header(t1)

Ensures:

- v1 is of sort schema(t1)[c]
- header(t2) is equal to header(t1)
- for all c' in header(t2),
  - if c' is equal to c then schema(t2)[c'] is equal to the sort of v2

  - otherwise, then schema(t2)[c'] is equal to schema(t1)[c']
- nrows(t2) is equal to nrows(t1)

Description Consumes a Table, a ColName representing a column name, and a transformation function and produces a new Table where the transformation function has been applied to all values in the named column.

```
> addLastName =
    function(name):
      concat(name, " Smith")
    end
> transformColumn(students, "name", addLastName)
| name          | age | favorite color |
| ------------- | --- | -------------- |
| "Bob Smith"   | 12  | "blue"         |
| "Alice Smith" | 17  | "green"        |
| "Eve Smith"   | 13  | "red"          |
> quizScoreToPassFail =
    function(score):
      if score <= 6:
        "fail"
      else:
        "pass"
      end
    end
> transformColumn(gradebook, "quiz1", quizScoreToPassFail)
| name    | age | quiz1  | quiz2 | midterm | quiz3 | quiz4 | final |
| ------- | --- | ------ | ----- | ------- | ----- | ----- | ----- |
| "Bob"   | 12  | "pass" | 9     | 77      | 7     | 9     | 87    |
| "Alice" | 17  | "fail" | 8     | 88      | 8     | 7     | 85    |
| "Eve"   | 13  | "pass" | 9     | 84      | 8     | 8     | 77    |
```

renameColumns :: t1:Table * ccs:Seq<ColName * ColName> -> t2:Table

Constraints
Let n be the length of ccs Let c11 … c1n be the first components of the elements of ccs and c21 … c2n be the second components.
Requires:

- c1i is in header(t1) for all i
- [c11 … c1n] has no duplicates
- concat(removeAll(header(t1), [c11 … c1n]), [c21 … c2n]) has no duplicates

Ensures:

- header(t2) is equal to header(t1) with all c1i replaced with c2i
- for all c in header(t2),
  - if c is equal to c2i for some i then schema(t2)[c2i] is equal to schema(t1)[c1i]
  - otherwise, schema(t2)[c] is equal to schema(t2)[c]
- nrows(t2) is equal to nrows(t1)

Description Updates column names. Each element of ccs specifies the old name and the new name.

```
> renameColumns(students, [("favorite color", "preferred color"), ("name",
  "first name")])
| first name | age | preferred color |
| ---------- | --- | --------------- |
| "Bob"      | 12  | "blue"          |
| "Alice"    | 17  | "green"         |
| "Eve"      | 13  | "red"           |
> renameColumns(gradebook, [("midterm", "final"), ("final", "midterm")])
| name    | age | quiz1 | quiz2 | final | quiz3 | quiz4 | midterm |
| ------- | --- | ----- | ----- | ----- | ----- | ----- | ------- |
| "Bob"   | 12  | 8     | 9     | 77    | 7     | 9     | 87      |
| "Alice" | 17  | 6     | 8     | 88    | 8     | 7     | 85      |
| "Eve"   | 13  | 7     | 9     | 84    | 8     | 8     | 77      |
```

find :: t:Table * r:Row -> n:Error<Number>

Constraints
Requires:

- for all c in header(r), c is in header(t)
- for all c in header(r), schema(r)[c] is equal to schema(t)[c]

Ensures:

- either n is equal to error("not found") or n is in range(nrows(t))

Description Find the index of the first row that matches r.

```
> find(students, [row: ("age", 13)])
2
> find(students, [row: ("age", 14)])
error("not found")
```

groupByRetentive :: t1:Table * c:ColName -> t2:Table

Constraints
Requires:

- c is in header(t1)
- schema(t1)[c] is a categorical sort

Ensures:

- header(t2) is equal to ["key", "groups"]
- schema(t2)["key"] is equal to schema(t1)[c]
- schema(t2)["groups"] is Table
- getColumn(t2, "key") has no duplicates
- for all t in getColumn(t2, "groups"), schema(t) is equal to schema(t1)
- nrows(t2) is equal to length(removeDuplicates(getColumn(t1, c)))

Description Categorizes rows of the input table into groups by the key of each row. The key is computed by accessing the named column.

```
> groupByRetentive(students, "favorite color")
| key     | groups                             |
| ------- | ---------------------------------- |
| "blue"  | | name    | age | favorite color | |
|         | | ------- | --- | -------------- | |
|         | | "Bob"   | 12  | "blue"         | |
| "green" | | name    | age | favorite color | |
|         | | ------- | --- | -------------- | |
|         | | "Alice" | 17  | "green"        | |
| "red"   | | name    | age | favorite color | |
|         | | ------- | --- | -------------- | |
|         | | "Eve"   | 13  | "red"          | |
```

```
> groupByRetentive(jellyAnon, "brown")
| key   | groups                                                                                  |
| ----- | --------------------------------------------------------------------------------------- |
| false | | get acne | red   | black | white | green | yellow | brown | orange | pink  | purple | |
|       | | -------- | ----- | ----- | ----- | ----- | ------ | ----- | ------ | ----- | ------ | |
|       | | true     | false | false | false | true  | false  | false | true   | false | false  | |
|       | | true     | false | true  | false | true  | true   | false | false  | false | false  | |
|       | | false    | false | false | false | true  | false  | false | false  | true  | false  | |
|       | | false    | false | false | false | false | true   | false | false  | false | false  | |
|       | | false    | false | false | false | false | true   | false | false  | true  | false  | |
|       | | true     | false | true  | false | false | false  | false | true   | true  | false  | |
|       | | false    | false | true  | false | false | false  | false | false  | true  | false  | |
|       | | true     | false | false | false | false | false  | false | true   | false | false  | |
| true  | | get acne | red   | black | white | green | yellow | brown | orange | pink  | purple | |
|       | | -------- | ----- | ----- | ----- | ----- | ------ | ----- | ------ | ----- | ------ | |
|       | | true     | false | false | false | false | false  | true  | true   | false | false  | |
|       | | false    | true  | false | false | false | true   | true  | false  | true  | false  | |
```

groupBySubtractive :: t1:Table * c:ColName -> t2:Table

Constraints
Requires:

- c is in header(t1)
- schema(t1)[c] is a categorical sort

Ensures:

- header(t2) is equal to ["key", "groups"]
- schema(t2)["key"] is equal to schema(t1)[c]
- schema(t2)["groups"] is Table
- getColumn(t2, "key") has no duplicates
- for all t in getColumn(t2, "groups"), header(t) is equal to removeAll(header(t1), [c])
- for all t in getColumn(t2, "groups"), schema(t) is a subsequence of schema(t1)
- nrows(t2) is equal to length(removeDuplicates(getColumn(t1, c)))

Description Similar to groupByRetentive but the named column is removed in the output.

```
> groupBySubtractive(students, "favorite color")
| key     | groups            |
| ------- | ----------------- |
| "blue"  | | name    | age | |
|         | | ------- | --- | |
|         | | "Bob"   | 12  | |
| "green" | | name    | age | |
|         | | ------- | --- | |
|         | | "Alice" | 17  | |
| "red"   | | name    | age | |
|         | | ------- | --- | |
|         | | "Eve"   | 13  | |
```

```
> groupBySubtractive(jellyAnon, "brown")
| key   | groups                                                                          |
| ----- | ------------------------------------------------------------------------------- |
| false | | get acne | red   | black | white | green | yellow | orange | pink  | purple | |
|       | | -------- | ----- | ----- | ----- | ----- | ------ | ------ | ----- | ------ | |
|       | | true     | false | false | false | true  | false  | true   | false | false  | |
|       | | true     | false | true  | false | true  | true   | false  | false | false  | |
|       | | false    | false | false | false | true  | false  | false  | true  | false  | |
|       | | false    | false | false | false | false | true   | false  | false | false  | |
|       | | false    | false | false | false | false | true   | false  | true  | false  | |
|       | | true     | false | true  | false | false | false  | true   | true  | false  | |
|       | | false    | false | true  | false | false | false  | false  | true  | false  | |
|       | | true     | false | false | false | false | false  | true   | false | false  | |
| true  | | get acne | red   | black | white | green | yellow | orange | pink  | purple | |
|       | | -------- | ----- | ----- | ----- | ----- | ------ | ------ | ----- | ------ | |
|       | | true     | false | false | false | false | false  | true   | false | false  | |
|       | | false    | true  | false | false | false | true   | false  | true  | false  | |
```

update :: t1:Table * f:(r1:Row -> r2:Row) -> t2:Table

Constraints
Requires:

- for all c in header(r2), c is in header(t1)

Ensures:

- schema(r1) is equal to schema(t1)
- header(t2) is equal to header(t1)
- for all c in header(t2)
  - if c in header(r2) then schema(t2)[c] is equal to schema(r2)[c]
  - otherwise, schema(t2)[c] is equal to schema(t1)[c]
- nrows(t2) is equal to nrows(t1)

Description Consumes an existing Table and produces a new Table with the named columns updated, using f to produce the values for those columns, once for each row.

```
> abstractAge =
    function(r):
      if (getValue(r, "age") <= 12):
        [row: ("age", "kid")]
      else if (getValue(r, "age") <= 19):
        [row: ("age", "teenager")]
      else:
        [row: ("age", "adult")]
      end
    end
> update(students, abstractAge)
| name    | age        | favorite color |
| ------- | ---------- | -------------- |
| "Bob"   | "kid"      | "blue"         |
| "Alice" | "teenager" | "green"        |
| "Eve"   | "teenager" | "red"          |
> abstractFinal =
    function(r):
      [row:
        ("midterm", 85 <= getValue(r, "midterm"))
        ("final", 85 <= getValue(r, "final"))]
    end
> update(gradebook, didWellInFinal)
| name    | age | quiz1 | quiz2 | midterm | quiz3 | quiz4 | final |
| ------- | --- | ----- | ----- | ------- | ----- | ----- | ----- |
| "Bob"   | 12  | 8     | 9     | false   | 7     | 9     | true  |
| "Alice" | 17  | 6     | 8     | true    | 8     | 7     | true  |
| "Eve"   | 13  | 7     | 9     | false   | 8     | 8     | false |
```

select :: t1:Table * f:(r1:Row * n:Number -> r2:Row) -> t2:Table

Constraints
Requires:

Ensures:

- schema(r1) is equal to schema(t1)
- n is in range(nrows(t1))
- schema(t2) is equal to schema(r2)
- nrows(t2) is equal to nrows(t1)

Description Projects each Row of a Table into a new Table.

```
> select(
    students,
    function(r, n):
      [row:
        ("ID", n),
        ("COLOR", getValue(r, "favorite color")),
        ("AGE", getValue(r, "age"))]
    end)
| ID | COLOR   | AGE |
| -- | ------- | --- |
| 0  | "blue"  | 12  |
| 1  | "green" | 17  |
| 2  | "red"   | 13  |
> select(
    gradebook,
    function(r, n):
      [row:
        ("full name", concat(getValue(r, "name"), " Smith")),
        ("(midterm + final) / 2", (getValue(r, "midterm")
        + getValue(r, "final")) / 2)]
    end)
| full name     | (midterm + final) / 2 |
| ------------- | --------------------- |
| "Bob Smith"   | 82                    |
| "Alice Smith" | 86.5                  |
| "Eve Smith"   | 80.5                  |
```

selectMany :: t1:Table * project:(r1:Row * n:Number -> t2:Table) * result:(r2:Row * r3:Row -> r4:Row) -> t2:Table

Constraints
Requires:

Ensures:

- schema(r1) is equal to schema(t1)
- n is in range(nrows(t1))
- schema(r2) is equal to schema(t1)
- schema(r3) is equal to schema(t2)
- schema(t2) is equal to schema(r4)

Description Projects each row of a table to a new table, flattens the resulting tables into one table, and invokes a result selector function on each row therein. The index of each source row is used in the intermediate projected form of that row.

```
> selectMany(
    students,
    function(r, n):
      if even(n):
        r
      else:
        head(r, 0)
      end
    end,
    function(r1, r2):
      r2
    end)
| name  | age | favorite color |
| ----- | --- | -------------- |
| "Bob" | 12  | "blue"         |
| "Eve" | 13  | "red"          |
> repeatRow =
    function(r, n):
      if n == 0:
        r
      else:
        addRows(repeatRow(r, n - 1), [r])
      end
    end
> selectMany(
    gradebook,
    repeatRow,
    function(r1, r2):
      selectColumns(r2, ["midterm"])
    end)
| midterm |
| ------- |
| 77      |
| 88      |
| 88      |
| 84      |
| 84      |
| 84      |
```

groupJoin<K> :: t1:Table * t2:Table * getKey1:(r1:Row -> k1:K) * getKey2:(r2:Row -> k2:K) * aggregate:(r3:Row * t3:Table -> r4:Row) -> t4:Table

Constraints
Requires:

Ensures:

- schema(r1) is equal to schema(t1)
- schema(r2) is equal to schema(t2)
- schema(r3) is equal to schema(t1)
- schema(t3) is equal to schema(t2)
- schema(t4) is equal to schema(r4)
- nrows(t4) is equal to nrows(t1)

Description Correlates the rows of two tables based on equality of keys and groups the results.

```
> getName =
    function(r):
      getValue(r, "name")
    end
> averageFinal =
    function(r, t):
      addColumn(r, "final", [average(getColumn(t, "final"))])
    end
> groupJoin(students, gradebook, getName, getName, averageFinal)
| name    | age | favorite color | final |
| ------- | --- | -------------- | ----- |
| "Bob"   | 12  | "blue"         | 87    |
| "Alice" | 17  | "green"        | 85    |
| "Eve"   | 13  | "red"          | 77    |
> nameLength =
    function(r):
      length(getValue(r, "name"))
    end
> tableNRows =
    function(r, t):
      addColumn(r, "nrows", [nrows(t)])
    end
> groupJoin(students, gradebook, nameLength, nameLength, tableNRows)
| name    | age | favorite color | nrows |
| ------- | --- | -------------- | ----- |
| "Bob"   | 12  | "blue"         | 2     |
| "Alice" | 17  | "green"        | 1     |
| "Eve"   | 13  | "red"          | 2     |
```

join<K> :: t1:Table * t2:Table * getKey1:(r1:Row -> k1:K) * getKey2:(r2:Row -> k2:K) * combine:(r3:Row * r4:Row -> r5:Row) -> t3:Table

Constraints
Requires:

Ensures:

- schema(r1) is equal to schema(t1)

- schema(r2) is equal to schema(t2)
- schema(r3) is equal to schema(t1)
- schema(r4) is equal to schema(t2)
- schema(t3) is equal to schema(r5)

Description Correlates the rows of two tables based on matching keys.

```
> getName =
    function(r):
      getValue(r, "name")
    end
> addGradeColumn =
    function(r1, r2):
      addColumn(r1, "grade", [getValue(r2, "final")])
    end
> join(students, gradebook, getName, getName, addGradeColumn)
| name    | age | favorite color | grade |
| ------- | --- | -------------- | ----- |
| "Bob"   | 12  | "blue"         | 87    |
| "Alice" | 17  | "green"        | 85    |
| "Eve"   | 13  | "red"          | 77    |
> nameLength =
    function(r):
      length(getValue(r, "name"))
    end
> join(students, gradebook, nameLength, nameLength, addGradeColumn)
| name    | age | favorite color | grade |
| ------- | --- | -------------- | ----- |
| "Bob"   | 12  | "blue"         | 87    |
| "Bob"   | 12  | "blue"         | 77    |
| "Alice" | 17  | "green"        | 85    |
| "Eve"   | 13  | "red"          | 87    |
| "Eve"   | 13  | "red"          | 77    |
```

B.4 Example Programs

This file challenges type systems with some programs that might be difficult to typecheck.

To keep the authenticity of some example programs, we assume the existence of the following functions in addition to the assumed function listed at the beginning of Table API document:

- fisherTest :: bs1:Seq<Boolean>, bs2:Seq<Boolean> -> n:Number, where the two sequences must be of the same length. This function performs the Fisher's exact test, and returns the p-value.
- sample<V> :: vs1:Seq<V> * n:Number -> vs2:Seq<V>, where n is in range(length(vs1) + 1).

B.4.1 dotProduct

This example defines a function that computes the dot-product of two numeric columns. When assigning a type to dotProduct, the type system should try to enforce that both c1 and c2 refer to numeric columns in t.

```
> dotProduct =
    function(t, c1, c2):
      ns = getColumn(t, c1)
      ms = getColumn(t, c1)
      sum(map(range(nrows(t)),
        function(i):
          ns[i] * ms[i]
        end))
    end
> dotProduct(gradebook, "quiz1", "quiz2")
183
```

B.4.2 sampleRows

This example defines a function that randomly samples rows of a table. This function might be interesting when working with tidy tables, where each row is one observation. "Pure" languages (e.g. Haskell) might find typing this example challenging because generating random number is stateful.

A type system should try to realize that sampleRows requires n is in range(nrows(t)) and ensures that the output table has the same schema as t and as many rows as n.

```
> sampleRows =
    function(t, n):
      indexes = sample(range(nrows(t)), n)
      selectRows(t, indexes)
    end
> sampleRows(gradebookMissing, 2)
| name    | age | quiz1 | quiz2 | midterm | quiz3 | quiz4 | final |
| ------- | --- | ----- | ----- | ------- | ----- | ----- | ----- |
| "Eve"   | 13  |       | 9     | 84      | 8     | 8     | 77    |
| "Alice" | 17  | 6     | 8     | 88      |       | 7     | 85    |
```

B.4.3 pHackingHomogeneous

Inspired by XKCD, this example program investigates the association between getting acne and consuming jelly beans of a particular color. The processed table, jellyAnon, is homogeneous because all of its columns contain boolean values. It is interesting to compare this program with the next example, pHackingHeterogeneous, which processes jellyNamed, a table that contains an additional string-typed column. Some type systems might understand this program but not the next one.

```
> pHacking =
    function(t):
      colAcne = getColumn(t, "get acne")
      jellyAnon = dropColumns(t, ["get acne"])
      for c in header(jellyAnon):
        colJB = getColumn(t, c)
        p = fisherTest(colAcne, colJB)
        if p < 0.05:
          println(
            "We found a link between " ++
            c ++ " jelly beans and acne (p < 0.05).")
        end
      end
> pHacking(jellyAnon)
We found a link between orange jelly beans and acne (p < 0.05).
```

B.4.4 pHackingHeterogeneous

This example program is similar to pHackingHomogeneous but processes a table with an extra column, "name". This column is dropped before calling the pHacking function. This example is interesting because the type system needs to understand that after dropping the column, the table contains only boolean values.

```
> pHacking(dropColumns(jellyNamed, ["name"]))
We found a link between orange jelly beans and acne (p < 0.05).
```

B.4.5 quizScoreFilter

This example computes the average quiz score for each student in gradebook. This example is interesting because the type system needs to understand the connection between the pattern of quiz column names (i.e. startsWith(..., "quiz")) and the type of those columns (i.e. numeric).

```
> buildColumn(
    gradebook,
    "average-quiz",
    function(row):
      quizColnames =
        filter(
          header(row),
          function(c):
            startsWith(c, "quiz")
          end)
      scores = map(
        quizColnames,
        function(c):
          getValue(row, c)
        end)
      sum(scores) / length(scores)
    end)
| name    | age | quiz1 | quiz2 | midterm | quiz3 | quiz4 | final | average-quiz |
| ------- | --- | ----- | ----- | ------- | ----- | ----- | ----- | ------------ |
| "Bob"   | 12  | 8     | 9     | 77      | 7     | 9     | 87    | 8.25         |
| "Alice" | 17  | 6     | 8     | 88      | 8     | 7     | 85    | 7.25         |
| "Eve"   | 13  | 7     | 9     | 84      | 8     | 8     | 77    | 8            |
```

B.4.6 quizScoreSelect

This example also computes the average quiz score for each student in gradebook. It computes quiz column names by concatenating "quiz" with numbers. This example is interesting because the type system needs to understand the connection between the computed column names and the type of those columns (i.e. numeric).

```
> quizColNames =
    map(
      range(4),
      function(i):
        concat("quiz", colNameOfNumber(i + 1))
      end)
> quizTable = selectColumns(gradebook, quizColNames)
> quizAndAverage =
    buildColumn(
      quizTable,
      "average",
      function(r):
        ns = map(header(r),
          function(c):
            getValue(r, c)
          end)
        average(ns)
      end)
> addColumn(
    gradebook,
    "average-quiz",
    getColumn(quizAndAverage, "average"))
```

```
| name    | age | quiz1 | quiz2 | midterm | quiz3 | quiz4 | final | average-quiz |
| ------- | --- | ----- | ----- | ------- | ----- | ----- | ----- | ------------ |
| "Bob"   | 12  | 8     | 9     | 77      | 7     | 9     | 87    | 8.25         |
| "Alice" | 17  | 6     | 8     | 88      | 8     | 7     | 85    | 7.25         |
| "Eve"   | 13  | 7     | 9     | 84      | 8     | 8     | 77    | 8            |
```

B.4.7 groupByRetentive

This example categorizes rows of the input table into groups based on the key in each row and does not drop the key column from the output table.

Ideally, this user-defined function should achieve the same type constraints as the version in the Table API.

```
> tableOfColumn =
    function(c, vs):
      t1 = addRows(emptyTable, map(vs, function(_): [row:] end))
      addColumn(t1, c, vs)
    end
> groupByRetentive =
    function(t, c):
      keys = tableOfColumn("key", removeDuplicates(getColumn(t, c)))
      makeGroup =
        function(kr):
          k = getValue(kr, "key")
          tfilter(t,
            function(r):
              getValue(r, c) == k
            end)
        end
      buildColumn(keys, "groups", makeGroup)
    end
```

B.4.8 groupBySubtractive

This example categorizes rows of the input table into groups based on the key in each row and drops the key column from the output table.

Ideally, this user-defined function should achieve the same type constraints as the version in the Table API.

```
> tableOfColumn =
    function(c, vs):
      t1 = addRows(emptyTable, map(vs, function(_): [row:] end))
      addColumn(t1, c, vs)
    end
> groupBySubtractive =
    function(t, c):
      keys = tableOfColumn("key", removeDuplicates(getColumn(t, c)))
      makeGroup =
        function(kr):
          k = getValue(kr, "key")
          g =
            tfilter(t,
              function(r):
                getValue(r, c) == k
              end)
          dropColumns(g, [c])
        end
      buildColumn(keys, "groups", makeGroup)
    end
```

B.5 Errors

This file presents a diagnostic challenge. Each example includes a buggy program and one or more corrected programs. A good programming media should help programmers to avoid writing these buggy programs or to recover from the bug and finally reach a corrected program.

These examples are adapted from student code collected in CS111 at Brown University.

To keep the authenticity of some error cases, we assume the existence of two plotting functions:

- scatterPlot :: t:Table * c1:ColName * c2:ColName -> Image, where both input columns must contain numbers.
- pieChart :: t:Table * c1:ColName * c2:ColName -> Image, where the first column must contain categorical values, and the second column must contain positive numbers.

B.5.1 Malformed Tables

This section lists errors that programmers can make when constructing table constants. All these malformed tables should be corrected to the students table, which is shown below with a full schema declaration.

```
| name    | age    | favorite color |
| String  | Number | String         |
| ------- | ------ | -------------- |
| "Bob"   | 12     | "blue"         |
| "Alice" | 17     | "green"        |
| "Eve"   | 13     | "red"          |
```

missingSchema This malformed table misses the schema.

```
| "Bob"   | 12 | "blue"  |
| "Alice" | 17 | "green" |
| "Eve"   | 13 | "red"   |
```

missingRow This malformed table misses the content of the last row. (*Note:* the last row is **not** a row with 3 missing values, but rather a row with no value.)

```
| name    | age    | favorite color |
| String  | Number | String         |
| ------- | ------ | -------------- |
| "Bob"   | 12     | "blue"         |
| "Alice" | 17     | "green"        |
|                                   |
```

missingCell The first row of this malformed table misses a cell.

```
| name    | age    | favorite color |
| String  | Number | String         |
| ------- | ------ | -------------- |
| "Bob"   | "blue" |
| "Alice" | 17     | "green"        |
| "Eve"   | 13     | "red"          |
```

swappedColumns The rows disagree with the schema on the ordering of the first two columns.

```
| name   | age     | favorite color |
| String | Number  | String         |
| ------ | ------- | -------------- |
| 12     | "Bob"   | "blue"         |
| 17     | "Alice" | "green"        |
| 13     | "Eve"   | "red"          |
```

schemaTooShort The schema specifies that there are two columns. But the rows have three columns.

```
| name    | age    |
| String  | Number |
| ------- | ------ |
| "Bob"   | 12     | "blue"         |
| "Alice" | 17     | "green"        |
| "Eve"   | 13     | "red"          |
```

schemaTooLong The schema specifies that there are four columns. But the rows have three columns.

```
| name    | age    | favorite number | favorite color |
| String  | Number | Number          | String         |
| ------- | ------ | --------------- |----------------|
| "Bob"   | 12     | "blue"          |
| "Alice" | 17     | "green"         |
| "Eve"   | 13     | "red"           |
```

B.5.2 Using Tables

This section lists errors in using tables. Each example comes with a context, which lists the used tables, and a task, which states how the table(s) should be used.

midFinal

Context gradebook

Task The programmer was asked to visualize as a scatter plot the connection between midterm and final exam grades.

A Buggy Program
```
> scatterPlot(gradebook, "mid", "final")
```

What is the Bug? The "mid" is not a valid column name of gradebook. However, the table contains a "midterm" column.

A Corrected Program
```
> scatterPlot(gradebook, "midterm", "final")
```

blackAndWhite

Context jellyAnon

Task The programmer was asked to build a column that indicates whether “a participant consumed black jelly beans and white ones”.

A Buggy Program

```
> eatBlackAndWhite =
    function(r):
      getValue(r, "black and white") == true
    end
> buildColumn(jellyAnon, "eat black and white", eatBlackAndWhite)
```

What is the Bug? The logical and appears at a wrong place. The task is asking the programmer to write getValue(r, "black") and getValue(r, "white"), but the buggy program accesses the invalid column "black and white" instead.

A Corrected Program

```
> eatBlackAndWhite =
    function(r):
      getValue(r, "black") and getValue(r, "white")
    end
> buildColumn(jellyAnon, "eat black and white", eatBlackAndWhite)
```

pieCount

Context jellyAnon

Task The programmer was asked to visualize the proportion of participants getting acne.

A Buggy Program

```
> showAcneProportions =
    function(t):
      pieChart(count(t, "get acne"), "true", "get acne")
    end
> showAcneProportions(jellyAnon)
```

What is the Bug? The program supplies a table produced by count to pieChart, which also expects a table and two of its column names. The table produced by count contains two column names, "value" and "count". Neither of the supplied colum names, "true" and "get acne", are column names of count(...).

A Corrected Program

```
> showAcneProportions =
    function(t):
      pieChart(count(t, "get acne"), "value", "count")
    end
> showAcneProportions(jellyAnon)
```

brownGetAcne

Context jellyNamed

Task The programmer was asked to compute how many participants consumed brown jelly beans and got acne, and how many did not.

A Buggy Program

```
> brownAndGetAcne =
    function(r):
      getValue(r, "brown") and getValue(r, "get acne")
    end
> brownAndGetAcneTable =
    buildColumn(jellyNamed, "part2", brownAndGetAcne)
> count(brownAndGetAcneTable, "brown and get acne")
```

What is the Bug? The built column was named inconsistently. In buildColumn(...), the column was named "part2" but when counted, the column was accessed with "brown and get acne".

A Corrected Program

```
> brownAndGetAcne =
    function(r):
      getValue(r, "brown") and getValue(r, "get acne")
    end
> brownAndGetAcneTable =
    buildColumn(jellyNamed, "brown and get acne", brownAndGetAcne)
> count(brownAndGetAcneTable, "brown and get acne")
```

getOnlyRow

Context students

Task The programmer was asked to find Alice's favorite color.

A Buggy Program

```
> getValue(
    getRow(
      tfilter(students,
        function(r):
          getValue(r, "name") == "Alice"
        end),
      1),
    "favorite color")
```

What is the Bug? There is only one row that matches the filtering criteria. So the only valid index is 0, not 1.

A Corrected Program

```
> getValue(
    getRow(
      tfilter(students,
        function(r):
          getValue(r, "name") == "Alice"
        end),
      0),
    "favorite color")
```

favoriteColor

Context students

Task The programmer was asked to define a function that finds all participants who like "green".

A Buggy Program

```
> participantsLikeGreen =
    function(t):
      tfilter(t,
        function(r):
          getValue(r, "favorite color")
        end)
    end
```

What is the Bug? The programmer returns getValue(r, "favorite color") directly in the predicate but should return a boolean.

A Corrected Program

```
> participantsLikeGreen =
    function(t):
      tfilter(t,
        function(r):
          getValue(r, "favorite color") == "green"
        end)
    end
```

brownJellybeans

Context jellyAnon

Task The programmer was asked to count the number of participants that consumed jelly beans of a given color.

A Buggy Program

```
> countParticipants =
    function(t, color):
      nrows(tfilter(t, keep))
    end
> keep =
    function(r):
      getValue(r, "color")
    end
> countParticipants(jellyAnon, "brown")
```

What is the Bug? "color" is not a valid column name. Instead of a string literal, the color should be a variable refering to the color defined in countParticipants.

A Corrected Program (1/2)

```
> countParticipants =
    function(t, color):
      nrows(tfilter(t, keep(color)))
    end
> keep =
    function(color):
      function(r):
        getValue(r, color)
      end
    end
> countParticipants(jellyAnon, "brown")
```

A Corrected Program (2/2)

```
> countParticipants =
    function(t, color):
      keep =
        function(r):
          getValue(r, color)
        end
      nrows(tfilter(t, keep))
    end
> countParticipants(jellyAnon, "brown")
```

employeeToDepartment

Context

- employees
- departments

Task The programmer was given two tables, one maps employee names to department IDs, the other maps department IDs to department names. The task is to define a function, employeeToDepartment that consumes the two tables and looks up the a department name that an employee corresponds to.

A Buggy Program

```
> lastNameToDeptId =
    function(deptTab, name):
      matchName =
        function(r):
          getValue(r, "Last Name") == name
        end
      matchedTab = tfilter(deptTab, matchName)
      matchedRow = getRow(matchedTab, 0)
      getValue(matchedRow, "Department ID")
    end
> employeeToDepartment =
    function(name, emplTab, deptTab):
      buildColumn(emplTab, "Department Name",
        function(r):
          lastNameToDeptId(deptTab, getValue(r, "Last Name"))
        end)
    end
```

What is the Bug? There are several problems in this program. First, employeeToDepartment is expected to return a department name, but it returns a table. Another problem is that the helper function is named lastNameToDeptId. The name suggests that this function maps the employee names to department IDs. But in employeeToDepartment, lastNameToDeptId is expected to produce department names. Finally, deptTab, the first parameter of lastNameToDeptId, has a name suggesting that it is bound to a department table. However, lastNameToDeptId uses deptTab as an employee table.

A Corrected Program

```
> deptIdToDeptName =
    function(deptTab, deptId):
      matchName =
        function(r):
          getValue(r, "Department ID") == deptId
        end
      matchedTab = tfilter(deptTab, matchName)
      matchedRow = getRow(matchedTab, 0)
      getValue(matchedRow, "Department Name")
    end
> employeeToDepartment =
    function(name, emplTab, deptTab):
      matchName =
        function(r):
          getValue(r, "Last Name") == name
        end
      matchedTab = tfilter(emplTab, matchName)
      matchedRow = getRow(matchedTab, 0)
      deptId = getValue(matchedRow, "Department ID")
      deptIdToDeptName(deptTab, deptId)
    end
```

References

[1] Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. "POPLMark reloaded: Mechanizing proofs by logical relations". In: *JFP* 29 (2019), e19.

[2] Leif Andersen, Michael Ballantyne, and Matthias Felleisen. "Adding interactive visual syntax to textual code". In: *PACMPL* 4.OOPSLA (2020), 222:1–222:28.

[3] Tudor Antoniu, Paul A. Steckler, Shriram Krishnamurthi, Erich Neuwirth, and Matthias Felleisen. "Validating the Unit Correctness of Spreadsheet Programs". In: *ICSE*. 2004, pages 439–448.

[4] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yellick. *The Landscape of Parallel Computing Research: A View from Berkeley*. Technical report UCB/EECS-2006-183. University of California at Berkeley, 2006.

[5] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. "Mechanized Metatheory for the Masses: The PoplMark Challenge". In: *TPHOLs*. 2005, pages 50–65.

[6] Titus Barik, Chris Parnin, and Emerson Murphy–Hill. "One λ at a time: What do we know about presenting human-friendly output from a program analysis tool?" In: *PLATEAU*. 2017.

[7] Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. "Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research". In: *ITiCSE-WGR*. 2019, pages 177–210.

[8] Alan F. Blackwell and Thomas R. G. Green. "Notational Systems – the Cognitive Dimensions of Notations framework". In: *HCI Models, Theories, and Frameworks: Toward a Multidisciplinary Science*. Morgan Kaufmann, 2003, pages 103–134.

[9] Bootstrap Community. *Bootstrap:Data Science*. Accessed 2021-05-26. URL: https://www.bootstrapworld.org/materials/data-science/.

[10] Peter Buneman and Atsushi Ohori. "Polymorphism and Type Inference in Database Programming". In: *ACM Trans. Database Syst.* 21.1 (1996), pages 30–76.

[11] Chris Chambers and Martin Erwig. "Reasoning about spreadsheets with labels and dimensions". In: *J. Vis. Lang. Comput.* 21.5 (2010), pages 249–262.

[12] Adam Chlipala. "Ur: statically-typed metaprogramming with type-level record computation". In: *PLDI*. 2010, pages 122–133.

[13] E. F. Codd. *The Relational Model for Database Management, Version 2*. Addison-Wesley, 1990.

[14] Will Crichton, Scott Kovach, and Gleb Shevchuk. *Expressiveness Benchmark*. Accessed 2021-09-12. URL: https://willcrichton.net/expressiveness-benchmark.

[15] C. J. Date. *Relational database writings: 1985-1989*. Addison-Wesley, 1990.

[16] LINQ developers. *Enumerable.Aggregate Method*. Accessed 2021-05-26. URL: https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.aggregate?view=net-5.0.

[17] MySQL developers. *MySQL 8.0 Reference Manual*. Accessed 2021-05-26. URL: https://dev.mysql.com/doc/refman/8.0/en/.

[18] Postgres developers. *Postgres 13.3 Documentation*. Accessed 2021-05-28. URL: https://www.postgresql.org/docs/13/index.html.

[19] R Developers. *R: The R Project for Statistical Computing*. Accessed 2021-05-26. URL: https://www.r-project.org.

[20] TypeScript Developers. *Keyof Type Operator*. Accessed 2021-05-25. URL: https://www.typescriptlang.org/docs/handbook/2/keyof-types.html.

[21] Jonathan Edwards. *Subtext: uncovering the simplicity of programming*. Accessed 2021-08-25. URL: http://www.subtext-lang.org/.

[22] Sebastian Erdweg et al. "Evaluating and comparing language workbenches: Existing results and benchmarks for the future". In: *Comput. Lang. Syst. Struct.* 44 (2015), pages 24–47.

[23] Benedict R. Gaster. "Records, variants and qualified types". PhD thesis. University of Nottingham, UK, 1998.

[24] Timnit Gebru, Jamie Morgenstern, Briana Vecchione, Jennifer Wortman Vaughan, Hanna M. Wallach, Hal Daumé III, and Kate Crawford. "Datasheets for Datasets". In: *CoRR* abs/1803.09010 (2018). URL: http://arxiv.org/abs/1803.09010.

[25] Charles A. E. Goodhart. "Problems of monetary management: the UK experience". In: *Monetary theory and practice*. Springer, 1984, pages 91–121.

[26] Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. "FERRY: database-supported program execution". In: *SIGMOD*. 2009, pages 1063–1066.

[27] Robert Harper and Benjamin C. Pierce. "A Record Calculus Based on Symmetric Concatenation". In: *POPL*. 1991, pages 131–142.

[28] Daniel Jackson. "Towards a theory of conceptual design for software". In: *Onward!* 2015, pages 282–296.

[29] C. B. Jay. *The FISh language definition*. Technical report. University of Technology, Sydney, 1998.

[30] Jsoftware, Inc. *The J Programming Language*. Accessed 2021-05-25. URL: https://www.jsoftware.com.

[31] JuliaData. *DataFrames.jl*. Accessed 2021-05-15. URL: https://github.com/JuliaData/DataFrames.jl.

[32] JuliaData. *DataTables.jl*. Accessed 2021-05-15. URL: https://github.com/JuliaData/DataTables.jl.

[33] Milod Kazerounian, Sankha Narayan Guria, Niki Vazou, Jeffrey S. Foster, and David Van Horn. "Type-level computations for Ruby libraries". In: *PLDI*. 2019, pages 966–979.

[34] Eugen Kiss. *7GUIs: A GUI Programming Benchmark*. Accessed 2021-09-12. URL: https://eugenkiss.github.io/7guis/more.

[35] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation". In: *SCAM*. 2009, pages 168–177.

[36] Clifford Konold, William Finzer, and Kosoom Kreetong. "Students' methods of recording and organizing data". In: *Annual Meeting of the American Educational Research Association*. 2014.

[37] Shriram Krishnamurthi and Kathi Fisler. "Data-Centricity: A Challenge and Opportunity for Computing Education". In: *Comm. ACM* 63.8 (2020), pages 24–26.

[38] John Launchbury and Simon L. Peyton Jones. "State in Haskell". In: *LISP Symb. Comput.* 8.4 (1995), pages 293–341.

[39] Benjamin S. Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. "Searching for type-error messages". In: *PLDI*. 2007, pages 425–434.

[40] John H. Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. "The Scratch Programming Language and Environment". In: *ACM Trans. Comput. Educ.* 10.4 (2010), 16:1–16:15.

[41] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. "Measuring the Effectiveness of Error Messages Designed for Novice Programmers". In: *ACM Technical Symposium on Computer Science Education*. 2011.

[42] Wes McKinney. "Data Structures for Statistical Computing in Python". In: *Proceedings of the 9th Python in Science Conference*. Edited by Stéfan van der Walt and Jarrod Millman. 2010, pages 56–61. DOI: 10.25080/Majora-92bf1922-00a.

[43] Erik Meijer. "The World According to LINQ". In: *Comm. ACM* 54.10 (2011), pages 45–51.

[44] J. Garrett Morris and James McKinna. "Abstracting extensible data types: or, rows by any other name". In: *PACMPL* 3.POPL (2019), 12:1–12:28.

[45] Marion R. Morrissett. "Missing Data in the Relational Model". PhD thesis. University of Virginia, 2013.

[46] Atsushi Ohori and Katsuhiro Ueno. "Making standard ML a practical database programming language". In: *ICFP*. 2011, pages 307–319.

[47] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. "Live functional programming with typed holes". In: *PACMPL* 3.POPL (2019), 14:1–14:32.

[48] Thomas Petricek. "Data Exploration Through Dot-Driven Development". In: *ECOOP*. 2017, 21:1–21:27.

[49] Tomas Petricek. "Foundations of a live data exploration environment". In: *Art Sci. Eng. Program.* 4.3(8) (2020), pages 1–37.

[50] Didier Rémy. "Typechecking Records and Variants in a Natural Extension of ML". In: *POPL*. 1989, pages 77–88.

[51] Eric L. Seidel, Ranjit Jhala, and Westley Weimer. "Dynamic witnesses for static type errors (or, Ill-Typed Programs Usually Go Wrong)". In: *JFP* 28 (2018), e13.

[52] Justin Slepak, Olin Shivers, and Panagiotis Manolios. "An Array-Oriented Language with Static Rank Polymorphism". In: *ESOP*. 2014, pages 27–46.

[53] Brown CS111 course staff. *CSCI 0111 Computing Foundations: Data*. Accessed 2021-05-26. URL: https://cs.brown.edu/courses/csci0111/.

[54] Charles Maurice Stebbins and Mary H. Coolidge. *Golden Treasury Readers: Primer*. Illustrator Unknown. American Book Co., 1909.

[55] Marilyn Strathern. "'Improving ratings': audit in the British University system". In: *European review* 5.3 (1997), pages 305–321.

[56] Don Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, and Tomas Petricek. "Themes in information-rich functional programming for internet-scale data sources". In: *DDFP*. 2013, pages 1–4.

[57] The pandas development team. *pandas-dev/pandas: Pandas*. Version latest. Feb. 2020. URL: https://doi.org/10.5281/zenodo.3509134.

[58] Tim Teitelbaum and Thomas W. Reps. "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment". In: *Comm. ACM* 24.9 (1981), pages 563–573.

[59] Tidyverse. *Tidyverse: R packages for data science*. Accessed 2021-05-26. URL: https://www.tidyverse.org.

[60] V. Javier Traver. "On Compiler Error Messages: What They Say and What They Mean". In: *Adv. in HCI* 2010 (2010), 602570:1–602570:26.

[61] Kai Trojahner and Clemens Grelck. "Dependently typed array programs don't go wrong". In: *J. Log. Alg. Meth. Program.* 78.7 (2009), pages 643–664.

[62] Niki Vazou, Alexander Bakst, and Ranjit Jhala. "Bounded refinement types". In: *ICFP*. 2015, pages 48–61.

[63] Walter G. Vincenti. *What Engineers Know and How They Know It*. John Hopkins University Press, 1993.

[64] Mitchell Wand. "Finding the Source of Type Errors". In: *POPL*. 1986, pages 38–43.

[65] Mitchell Wand. "Type Inference for Record Concatenation and Multiple Inheritance". In: *LICS*. 1989, pages 92–97.

[66] Mitchell Wand. "Type Inference for Record Concatenation and Multiple Inheritance". In: *Inf. Comput.* 93.1 (1991), pages 1–15.

[67] Hadley Wickham. *Advanced R*. Chapman and Hall/CRC, 2014.

[68] Hadley Wickham. "Tidy data". In: *Journal of statistical software* 59.1 (2014), pages 1–23.

[69] Jack Williams, Carina Negreanu, Andrew D. Gordon, and Advait Sarkar. "Understanding and Inferring Units in Spreadsheets". In: *Visual Languages and Human Centric Computing*. 2020, pages 1–9.

[70] Michael S. Wogalter, Vincent C. Conzola, and Tonya L. Smith-Jackson. "Research-based guidelines for warning design and evaluation". In: *Applied Ergonomics* 33 (2002).

[71] John Wrenn and Shriram Krishnamurthi. "Error Messages Are Classifiers: A Process to Design and Evaluate Error Messages". In: *SPLASH Onward!* 2017.

[72] xkcd. *Significant*. Accessed 2021-05-13. URL: https://xkcd.com/882.

[73] Sherry Yang, Margaret M. Burnett, Elyon DeKoven, and Moshé M. Zloof. "Representation Design Benchmarks: A Design-Time Aid for VPL Navigable Static Representations". In: *J. Vis. Lang. Comput.* 8.5-6 (1997), pages 563–599.

About the authors

Kuang-Chen Lu (LuKuangchen1024@gmail.com) is a PhD student at Brown University.

Ben Greenman (benjamin.l.greenman@gmail.com) is a PLT member and a postdoc at Brown University.

Shriram Krishnamurthi (shriram@brown.edu) is the Vice President of Programming Languages (no, not really) at Brown University.