RSheet

One of the earliest "killer apps" for a computer was VisiCalc. Released on the Apple II in 1979, it was the first spreadsheet program, and it was a huge success. It was so successful that it was the main reason that people bought Apple II computers. In this assignment, you will be implementing a simple spreadsheet program in Rust.

Specifically, you will build a server program which is controlled through simple commands. Those commands will allow you to set and get values within a spreadsheet. Since values in the spreadsheet may be dependent on other values, you will also need to manage cells being calculated and re-calculated.

The goals of this assignment are:

1. To get experience with Rust's constructs for managing concurrency.
2. To design and structure a program capable of correctly and (somewhat) efficiently managing concurrent interactions.
3. To have fun creating an aesthetic and interesting application.

We want to also be explicit about what the goals aren't:

1. To assess your ability to write spreadsheet formulae. Specifically, while we use a particular language to write formulae, you don't need to learn to use it yourself.
2. To assess your ability to write large amounts of useless cruft solely for us to mark you on. Where you're writing code, we have a reason for it. Where you're writing text, it's because we want to genuinely understand your thinking.

Part of this assignment involves submitting a mark\_request.txt file. This gives you an opportunity to talk about both the highlights and limitations of your design. If you can identify limitations in your design, we'll give you some marks even where we would have otherwise taken them away, since identifying sub-optimal design and learning from it is part of the challenge of the assignment.

An Introduction to Spreadsheets

A spreadsheet is a grid of cells, each of which can contain a value. Cells are addressed using the "A1" method. Each cell has a column and a row. The characters in the column are the column name, and the number is the row number. For example, the cell in the first column and the first row is "A1". The cell in the second column and the first row is "B1". The cell in the first column and the second row is "A2". Importantly, the cell in the 27th column and the 1st row is "AA1". (after AA comes AB, AC, and so on until BA...). Because it is surprisingly difficult to write this logic yourself; we have provided a function called [column\_number\_to\_name](https://docs.rs/rsheet_lib/latest/rsheet_lib/cells/fn.column_number_to_name.html)to do this conversion for you. To avoid integer overflow, you will never be asked to write to a cell below or to the right of ZZZZZZ999999

In this program, the logic for spreadsheets will be executed using the [Rhai Language](https://github.com/rhaiscript/rhai/). Rhai is a small, fast, safe and embeddable scripting language that feels very similar to Rust. We have provided a type [CellExpr](https://docs.rs/rsheet_lib/latest/rsheet_lib/cell_expr/struct.CellExpr.html)which manages your interactions with this language for you. You do not need to understand the Rhai language, nor know how to use it. You will only need to use the simple CellExpr type in your assignment. You must not parse the Rhai code yourself: you should simply pass it to the CellExpr.

How your program will work

[We have provided you with starter code.](https://cgi.cse.unsw.edu.au/~cs6991/25T1/assignment/02/starter.tar) You **must** use this code as a starting point for your assignment. The starter code manages all the input and output for you, such that you only need to implement the rsheet library. That is, you will not have to modify main.rs. The starter code explains exactly what you will need to do.

To use the starter code, you have two options:

* To start with, you can run the starter code with no arguments. This will let you enter commands, and see what effect they have. This is the easiest way to run your code, and is generally recommended.
* Alternatively, you can run the starter code with a single argument, which is a network address. This will cause your program to start receiving instructions over a network connection. Running 6991 rsc <address> will allow you to connect to your program over a command line. This can make it easy to test multiple connections at once, or to use other interfaces to your program. You can use an address like 127.0.0.1:6991, which means "on my local computer, on port 6991". If you are on CSE (which is a shared machine with many students), there may already be someone using port 6991, so come up with a random number instead (less than 65535).

We have provided a reference solution to check behaviour you are unsure of. You can run the reference solution with the command: 6991 rsheet.

How We'll Mark Your Code

In this assignment, we won't be providing Design Excellence like in Assignment 1. That's mainly because apart from just "add more features", there wasn't really anything we felt would significantly add to the experience of doing this assignment.

Markers also won't be providing general feedback over your whole assignment. We've done this for a few reasons:

* We've already had an opportunity to give general Rust feedback to you on Assignment 1
* This assignment focusses more on concurrent design than on general design.
* Necessarily, feedback for this assignment will be given after the end of term so we often find students aren't as interested in general feedback.

Instead, we'll be asking you to answer 5 questions to reflect on your design. In that, you'll point out some particular areas of your code; and where necessary we'll leave feedback on those answers and those bits of code.

Furthermore, if you have particular questions about your design; we encourage you to leave them in the mark\_request.txt file. Your marker will reply to those questions. Notably, you only need to leave questions if you want to. Also importantly, we've asked markers to be as specific with their answers as you are with your questions. For example, "how could I have done better?" is a pretty general question, we'll give some general answers. "in my Blah struct I returned a `Box`, is there a better way to do that?" is a much more specific question.

The Tasks To Complete

Stage 1. Basic Get and Set (15%)

The basis of this assignment is two commands: get and set. These allow you to interact with the state of the spreadsheet. You can use the Manager trait to receive these commands by calling the Manager::accept\_new\_connection function, which can be called repeatedly to accept new connections. Once Connect::NoMoreConnections is received, the server will shutdown/terminate once all existing connections are closed. A connection is made up of a reader and a writer. You will read messages (the get and set input commands) from the reader, and write your responses to the writer as a [rsheet\_lib::replies::Reply](https://docs.rs/rsheet_lib/latest/rsheet_lib/replies/enum.Reply.html), which is an enum with either a cell's value, or an error. A connection is considered closed once a ReadMessageResult::ConnectionClosed or WriteMessageResult::ConnectionClosed is received from read\_message or write\_message. Any errors returned from those two functions can be considered fatal/unrecoverable, i.e. you may terminate your program and output that error. In practice (i.e. in our tests), these unrecoverable errors should never occur.

For now, you will only have one user talking to your code, so you do not need to handle multiple connections.

We have provided you with the [rsheet\_lib::command::Command](https://docs.rs/rsheet_lib/latest/rsheet_lib/command/enum.Command.html) type, which implements the FromStr trait, so you can use it for parsing the get and set commands (a demonstration on how to do this is provided in the starter code).

The get command takes one argument: a cell name/identifier. It returns the value of that cell. By default, all cells are CellValue::None until modified. Cells can have one of four types of values: None (the default), an integer (i64), a string (String), or an error (also represented as a String).

The set command takes the name of a cell, and then a string (which might contain many spaces!), which is an expression that evaluates to a value. It sets the cell to this value. The set command has no output. That said, the set command *must occur atomically*. In other words, if you receive a set command, then a get command; the set command must finish before the get command starts.

For the avoidance of doubt, you **must** evaluate the expression given to you when you receive a set command. Some people have suggested implementations that wait until a get happens, and only evaluates the Rhai cell expression then... this is not allowed..

Given a set command you can use the [rsheet\_lib::cell\_expr::CellExpr](https://docs.rs/rsheet_lib/latest/rsheet_lib/cell_expr/struct.CellExpr.html) struct, and its evaluate method to evaluate the expression. This will take the expression, and evaluate it correctly; returning a CellValue.

For example, it will take 2 + (4 \* 6) and evaluate that to 26.

A note on the spreadsheet language

The spreadsheet formula language is [Rhai](https://rhai.rs/). You do not need to learn any Rhai to be able to complete this assignment, but if you're interested, it's a Rust-like scripting language. All the usual arithmetic operations are supported, as well as string concatenation ("abc" + "def" == "abcdef").

We've also implemented two useful built-in functions for you:

* sum(arg) takes a list or list of lists; and returns the sum.
* sleep\_then(time, value) takes a time (in ms) and a value. It waits the given number of milliseconds, then executes.

Error Handling

* If the CellExpr::evaluate function encounters an error while trying to evaluate the cell expression (e.g. it tried to add a string and a number, which is not supported in Rhai), it will return an Ok(CellValue::Error). You should treat this CellValue::Error like any other cell value. However, if any of the variable values provided to CellExpr::evaluate are CellValue::Errors, CellExpr::evaluate will return Err(CellExprEvalError), which means that one of the dependency cells is an error. During the set (because we evaluate on set not get), you must figure out a way to represent and store this additional error state of a cell, namely that cell e.g. A1 depends on another cell with an error. Then, if the user tries to get the value of A1, you need to return a Reply::Error stating that getting the value of a cell that depends on another cell with an error is not allowed.
* If calling the parse::() function returns a parsing error, which could happen if you receive an invalid command (e.g. frobnicate B1) or an invalid cell reference (e.g. get r2d2), then you should return a Reply::Error with that error.

Examples

A simple example:

**get A1**

A1 = None

**set A1 3 + 5**

**get A1**

A1 = 8

A more complex example:

**set ASDF a**

Error: Invalid Key Provided

**set A1 this-isn't-code**

**get A1**

A1 = Error: "'this' can only be used in functions (line 1, position 1)"

A note on --mark-mode

The starter code includes a variable, --mark-mode. In this mode, any errors you send back to the user are replaced with a standard error message. You should write descriptive error messages in your program, and they can be completely different to the reference solution. During marking, only the fact you've sent back an error message will be checked, not the contents of the error.

Stage 2. Using Variables in Calculations (15%)

A spreadsheet isn't very useful unless it can actually use variables. In this stage, you will add support for variables into rsheet.

Variables can be used in the set command simply by referencing them. For example, set A2 A1 + 1 means "set the A2 cell to be equal to A1 + 1". To make this work, you should use the [cell\_expr::CellExpr::find\_variable\_names](https://docs.rs/rsheet_lib/latest/rsheet_lib/cell_expr/struct.CellExpr.html" \l "method.find_variable_names)function to find the names of variables referenced inside a command, and then pass them to the [cell\_expr::CellExpr::evaluate](https://docs.rs/rsheet_lib/latest/rsheet_lib/cell_expr/struct.CellExpr.html" \l "method.evaluate)function.

An important note, **which only applies until Stage 4** (at which point you will lose this guarantee) is that you can assume that the cells on which other cells depend will never change.

**set A1 1**

**set A2 A1 + 1**

**get A2**

A2 = 2

**set A1 3** # <---- this will never happen

**get A2** # <---- this will never happen

A2 = 4

This means you don't need to handle the dependencies changing, and updating them until stage 5.

There are three types of variables you should support:

* *Scalar Variables*: these are individual cells; like A1. They will contain a single value, either a string or an integer.
* *Vector Variables*: these are a vertical or horizontal list of cells; spelled A1\_A3 or A1\_C1. These are represented by a 1-dimensional list.
* *Matrix Variables*: these are rectangular selections of cells; like A1\_B3. They will contain a list of lists; where each sub-list corresponds to a **row** (not a column) in the spreadsheet.

The main place we will use these vector and matrix variables is with the sum command that we've built into the spreadsheet language (i.e. Rhai)

**set A1 1**

**set A1 1**

**set A2 1**

**set A3 sum(A1\_A2)**

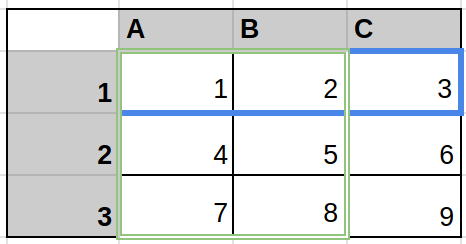
**get A3**

A3 = 2

You will always receive the top-left variable and the bottom-right variable in a vector or matrix, in that order. This means you'll always get A1\_B3 and never B3\_A1 or B1\_A3. Similarly, you'll always get A1\_C1 and never C1\_A1 or A3\_A1. You don't need to test for this case.

You will never receive vector / matrix variables that contain exactly one value, like A1\_A1 or B3\_B3. You do not need to test for this case.

Here is an example of what certain variables would equate to if the sheet was:



In the image, the blue outline is A1\_C1 and the green outline is A1\_B3. These would be represented as [1, 2, 3] and [[1, 2], [4, 5], [7, 8]] respectively.

Stage 3. Multiple Readers and Writers (20%)

In this section, you will support more than one reader/writer accessing your spreadsheet simultaneously. Because you're using a terminal to interact with the spreadsheet, we have a special syntax that will allow you to pretend to be from multiple clients at once. This syntax is *already implemented*, you do not need to do anything to get it to work.

To use this special syntax, you'll do something like this:

**snd1: get A1**

A1 = None

**snd2: set A2 42**

**snd2: get A2**

A2 = 42

The part before the : uniquely indicates a "sender". If you type a new sender, a new connection will be made to your sheet. If you reuse an existing sender, it will send another command over the same connection.

To clarify, when you type something into the terminal, rsheet\_lib reads the line. It will split the line by the first colon, and then use the part before the colon as the "sender". If no colon is found, the sender will just be the empty string. If the sender has never been seen before, a new "connection" will be made to the sheet. If the sender has been seen before, the command will be sent over the existing connection.

In order to correctly implement this assignment, you will need to implement it such that each connection to the sheet is processed in its own thread. This is the first stage that requires multithreading.

One important part of this assignment is that interactions with the sheet from a single observer must happen in order. For example, from snd2's point of view, the set must happen before the get.

Of course, with multiple readers and writers, it's not guaranteed that two different senders's actions will happen in any particular order. To deal with this in testing, we've added a utility for marking called sleep, which ensures a gap of a certain number of milliseconds between commands. Thus:

**snd1: set A1 1**

**snd2: get A1**

A1 = ???

isn't guaranteed to say that A1 is set; but the following will.

**snd1: set A1 1**

**snd2: sleep 50** # <--- this is implemented for you as part of rsheet\_lib.

**snd2: get A1**

A1 = 1

We've designed all the test cases so that there should be one correct ordering of outputs. There are many other inputs to your program that could have multiple correct orderings, depending on exactly how the threads behave. For example, the following code block could set A1 to 1 or 2.

**snd1: set A1 1**

**snd2: set A1 2**

**snd1: get A1**

A1 = ???

We will not test you on any of these cases.

Stage 4. Simple Dependency Changes (20%)

In this stage, you will start to deal with simple dependencies. This means, for example, that you'll set B1 A1 \* 2 (i.e. B1 = (A1 \* 2)). Then, we might change A1. If A1 is set to 3, then B1 must be set to 6. If A1 is set to "blah", then B1 will be an error (since "blah" \* 2 causes an error in Rhai).

It is important that these dependency changes happen *asynchronously*. In other words, say that you've run the following commands:

**set A1 1**

**set B1 A1 + 1**

Because of the guarantee that sets occur atomically, you're assured that A1 will be equal to 1 by the time that we set B1. However, let's say you now run set A1 2. A1 will be equal to 2 immediately. However, B1 will not yet be equal to 3. That should happen separately, and not necessarily atomically.

Specifically, you **must** have a single worker thread which processes all the updates to other cells in the spreadsheet that are not directly caused by an update (i.e. set) command. This worker thread will call the CellExpr::evaluate function for each cell whose dependency has been updated.

Importantly, because these updates are happening in the background on the worker thread, this code is not guaranteed to have B1 be 3 straight away, since the worked thread may need some time to process.

**set A1 1**

**set B1 A1 + 1**

**set A1 2**

**get B1** # <-- potentially B1 is still 2

For that reason, we'll usually put a sleep that gives your extra thread enough time to process B1 before we check it.

It's also important to note that for this section, we will *only* test dependency chains exactly one-cell long. You'll never end up in a situation where C1 depends on B1, and B1 depends on A1 (in this stage). You'll also never be given a circular dependency or self-referential dependency (in this stage).

Note that if you receive a command which gets a cell which depends on an error, you should return a Reply::Error. For example, if A1 is set to bad-code, and B1 is set to A1 + 1, then get B1 should fail with an error stating that it depends on a cell with an error.

A Complex Edge Case

As part of completing this stage, there is a complex edge case which you will need to decide how to handle. This edge case involves the fact that dependencies can be updated in strange orders.

In particular, some cell updates might take some time (emulated using the sleep\_then function). Say user A makes an update to a cell that takes 5 seconds to compute. User B then makes an update to that same cell 1 second later, but that update only takes 2 seconds to compute. In rsheet, we want to make sure a more recent update is never wiped out by an older one. Since user B's update was more recent, once user A's update finishes computing (at T=5 seconds), it **should not** override the value produced by user B.

Here is an example in rsheet code:

**snd1: set B1 A1 + 1**

**snd1: set A1 sleep\_then(5000, 5)**

**snd2: sleep 1000**

**snd2: set A1 sleep\_then(2000, 10)**

The sleep\_then function can be located in the rsheet\_lib::cell\_expr module inside the evaluate function.

It's a simple function that sleeps for a given number of milliseconds, then returns the value given to it.

It's clear that A1 should be equal to 10 here (since it was the more recent update). That means that B1 should be equal to 11. However, this could be (incorrectly) written out like this:

* 0 seconds in: B1 got set,
* 0 seconds in: snd1 sleeps for 5 seconds, waiting to update A1
* 0 seconds in: snd2 sleeps for 1 second.
* 1 second in: snd2 sleeps for 2 seconds, waiting to update A1
* 3 seconds in: snd2 updates A1 = 10
* 3.001 seconds in: B1 updates to 11
* 5 seconds in: A1 is (incorrectly!) updated to 5
* 5.001 seconds in: B1 is (incorrectly) updates to 6.

Even though the user more recently set A1 to 10, it gets set *back* to 5, because it was slower to compute. Furthermore, any dependencies are also incorrectly set as a result of this mistake.

You will need to account for this edge case in your code.

Stage 5. Multi-Layered Dependencies (30%)

In this stage, you will expand your implementation of Stage 5 so that it works for dependency chains of any length. In other words, you might have a situation where C1 depends on B1, and B1 depends on A1. You might also have tricker dependencies, like where C1 depends on A1\_B1, and B1 depends on A1.

You are guaranteed that you will never be asked to calculate a circular dependency, or self-referential dependency. That is, there will never be a chain of dependencies such that a cell depends on itself, like set A2 A2 or set A1 A2; set A2 A1.

Autotests