Table Processor

Did you spot any mistakes? Do you have any questions? Please contact Filip Říha (rihafili) via MS Teams.

Fork the Table Processor repository (https://gitlab.fit.cvut.cz/BI-OOP/B231/table-processor) to start working on the project. **Do not forget to make your repo private!**

The assignment specification is currently prone to change, but the overall idea of the project is the same. If anything is unclear or not explained enough, do not hesitate to contact me (Filip Říha) for clarification. Last change was **29.11.2023**, full page change history here (https://gitlab.fit.cvut.cz/BI-OOP/bi-oop/-/commits/master/projects/table-processor.md) .

The main goal of the project is to create an application to process tables with formulas. We should be able to load the table, evaluate all formulas found in the table, filter the rows by their values in specified columns, and finally print the result.

Application Overview

1. Load an input file
2. Evaluate all formulas
3. Filter rows according to a filter
4. Select the desired range to print
5. Add header row and column if specified
6. Output the table in the desired format

Grading

The whole project is split into individual sections. Each section is evaluated separately with its own requirements and points. To get the full amount of points for the section, students must not only satisfy the requirements but also use objects and object modeling. The use of polymorphism, immutable objects, and design patterns is expected, but not required. The student should be able to justify and defend their design decisions.

The design of the application and its code must be clean, clear, and easy to understand. Do not over-engineer, unnecessarily complicated, or hard-to-read code may not receive the full amount of points.

Clean code and quality OOP design are valued over performance. The algorithms don’t have to be perfect.

Testing is a big part of any project, including this one. They are expected to be of the same standard as the rest of the code. **Hard to test code is the first sign of a bad design.** Tests should be automated and fast so that students can run them often. Sections of the project that are not properly tested (poor test coverage, badly designed tests, failing tests) can get up to a 50% point penalty.

Bonus features will have minimal to no impact on the final grade. The focus should be on the core of the requirements, clean code, quality OOP design, and good testing.

Libraries and dependencies

It is not allowed to use external libraries, except for testing.

Requirements

TL;DR

Requirement	Points
Load and parse	5
Evaluate the formulas	13
Pretty print	9
Filter on column	7
Select range	4
Output	6
CLI	8
Sum	**52**

Load and parse the table - 5 points

Load a table from a file in a CSV format (https://cs.wikipedia.org/wiki/CSV). Usually by default the CSV format uses a comma for separating columns, but other separators are also not uncommon (typically a tab, space, or semicolon). With the `--separator [SEPARATOR]` argument, the user can define what column separator should be used. When no separator is specified, the default comma (`,`) is used. The lines are separated by a newline character (`\n`). It can be expected that each row contains the same count of columns.

The individual cells of the table have three types:

- **Empty** (no value)
- **Number** (only integers)
- **Formula** (more in the next part)

The input file should be passed in by the argument `--input-file [FILE]` .

The loading must be extensible, so both the format can be changed (XML, HTML table) and the source can be different (network data, standard input).

Note: You don't need to worry about strings, quoting of the CSV cells, and string escaping (it's not a course about parsing), simple splitting the input file into lines, and then splitting by the column separator should be enough.

Evaluate the formulas - 13 points

The formula is denoted by starting with the equals sign (`=`) (similar to Excel formulas).

The operands are either **numbers**, or **references to other cells**. These are referred to by a location in the table as a letter denoting the column and a number denoting the row, also similar to an Excel cell. The rows start indexing at 1.

Illustration of the reference naming scheme:

	A	B	C	…
1	A1	B1	C1	…
2	A2	B2	C2	…
3	A3	B3	C3	…
…	…	…	…	…

The **operators** are `+`, `-`, `*`, and `/`, which execute the expected arithmetical operation (add, subtract, multiply, and divide). You don't need to handle operators' precedence, i.e., evaluate from left to right.

Extra 2 points can be gain for implementing proper operator precedence including support for parenthesis.

Additional operators must be easily added if needed.

Tip: It might be beneficial to first construct an Abstract Syntax Tree (https://en.wikipedia.org/wiki/Abstract_syntax_tree) for the given formula and then evaluate it.

Another tip: Beware of regular expressions. When used extensively, they can lead to non-extensible and/or hardly understandable code, as their composition can be tricky or can even result in incorrect code behavior. They are not forbidden, but they are discouraged.

Evaluation errors

If an **empty** or **error cell** would be evaluated in an arithmetical operation, this results in an error. A plain reference to an empty cell should result in an empty cell.

The evaluator also needs to check for **cyclical dependencies** between nodes (e.g. A1 references B1 and B1 references A1), also resulting in error when encountered.

If an error occurs, it should be reported to the user (some error message) and the evaluation either fails (without printing the output) or the cell is rendered as some error text (for example `<ERROR>`), this is up to the student.

The error cannot be silently ignored (for example rendering the error cell as empty).

Example

The input

```
1, 2
3, =A1+A2*2
```

results in the table

1	**2**
3	8

Pretty print the table - 9 points

The evaluated formula should be then translated into a text representation.

The minimum requirement is **two different output formats:**

- **CSV**
- **Markdown**

The format is specified by the argument `--format [TYPE]`, where the type is either `csv` or `md` (for markdown). It is up to the student which one is the default (when no format is passed).

When `csv` is selected, with the `--output-separator [SEPARATOR]` parameter the user can change the CSV column separator. By default it should use the comma (`,`).

The pretty printing must be easily extensible so that other formats can be easily added (HTML table, XML, etc.)

The CSV must be a correct CSV format (reference for CSV (https://en.wikipedia.org/wiki/Comma-separated_values#Specification)) and the Markdown table should be a correct Markdown (reference for Markdown tables (https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet#tables)).

Some other options can be specified which change the format of the output:

Headers

By specifying the `--headers` command-line option, the pretty printer should add a header row with the column names and row numbers.

Columns are named A, B, etc., and rows are indexed from the number 1, same as specified in the evaluation chapter. More can be seen in the example.

Filter on column value - 7 points

The individual rows of the table can be filtered by a predicate on a column. Each row is passed to the filter and if the value on the specified column does not satisfy the predicate, the row is not printed. There can be any number of filters specified.

This does not shift the indices of the rows. So for example, if we have non-empty rows 1 and 2 and row 1 gets filtered out, the remaining row still has index 2.

The filtering must be easily extensible to do additional types of filtering.

All of the specified filters must be implemented to get all points for this section.

Value filtering

These have the syntax of `--filter [COLUMN] [OPERATOR] [VALUE]`.

Column specifies the column in which the values are tested.

The **operator** is one of `<`, `>`, `>=`, `<=`, `==`, `!=`, performing the (in)equality check as is standard for C-like programming languages.

Value is a number against which the column is tested.

For example, `--filter C >= 15` prints only the rows that have in column C a number 15 or higher.

Empty cells do not satisfy this filter.

Empty cell filtering

The parameter `--filter-is-not-empty [COLUMN]` discards from the output any row which has an empty cell on the specified column.

Conversely, `--filter-is-empty [COLUMN]` filters out rows with any value on the specified column.

Print only the specified range - 4 points

The user can specify which range of the table to print out. The format of the argument is `--range [FROM] [TO]`. By passing this argument, the table prints all of the cells in a rectangle with the `FROM` and `TO` cells as in the corners.

For example, `--range B2 D4` prints columns B, C, and D on rows 2, 3, and 4.

If no range is specified, the table prints all cells with all empty rows stripped from the bottom and empty columns stripped from the right.

Pretty print examples

▼ *(collapsed for brevity)*

Example input

```
1,100,=A1+B1
2,1020,20
```

Example CSV output (no headers)

```
1,100,101
2,1020,20
```

Example CSV output (with headers)

```
,A,B,C
1,1,100,101
2,2,1020,20
```

Example Markdown output (no headers)

```
| 1 | 100 | 101 |
|---|---|---|
| 2 | 1020 | 20 |
```

which gets rendered as:

1	**100**	**101**
2	1020	20

Example Markdown output (with headers)

```
| | A | B | C
|---|---|---|---|
| 1 | 1 | 100 | 101 |
| 2 | 2 | 1020 | 20 |
```

which gets rendered as:

	A	**B**	**C**
1	1	100	101
2	2	1020	20

Output destination - 6 points

The user can specify an output file with the argument `--output-file [FILE]` to output the table to. When no file is specified, or the `--stdout` argument is used, print the table to the standard output.

CLI interface - 8 points

The individual parameters are specified in the individual sections. The ordering of the parameters is not specified and it should not be enforced. The only mandatory parameter is the **input file**, if it is missing, inform the user and exit.

It is a convention that if an incorrect command line argument is passed or the `--help` or `-h` flags are passed, the **help page** should be printed with the information about all of the available parameters and flags.

DO NOT have this help page as a single hardcoded string in your code, instead try to construct it from the individual parameter parsers.

You do not need to set the exit code on an unsuccessful execution.

The CLI must be also done in an OOP way. You do not need to use some specific pattern (MVC, Unidirectional Data Flow, etc.), but it has to be easily extensible with new parameters.

Testing - 50% of all points

Testing is an important part of the project. All features and modules need to be tested with unit tests. **If the code is not tested enough, up to 50% of the points can be lost.**

Do not test just the happy path, test edge cases, invalid inputs, or failing paths.

If you want to use the ScalaCheck (https://scalacheck.org/) library, feel free to do so, but it is not mandatory.

System tests are not discouraged, but they are optional and do not grant additional points.

Some quick tips and tricks

- do it in the OOP way, leverage polymorphism, avoid static methods, global state and singleton classes
- use case classes and pattern matching where appropriate
- use static methods only for some auxiliary functions and only when necessary
- avoid null, use Option or the null object pattern for possibly missing data
- think of extensibility - how can I add more input / output formats, more formula expressions, more filters
- add comments to all classes and their non-trivial methods
- make sure that you CLI and application code are separate and not coupled together

Table Processor
projects/table-processor.md, last change 9936b91f (2023-11-29 at 14:53, Filip Riha)