Assignment 7: Stanford 1-2-3

Excellent assignment by Julie Zelenski, with minor revisions by Jerry.

This is your last CS106X assignment! It is a chance to pull together your stellar C++ skills, design a complicated data structure, use a variety of existing classes, design and implement a few new ones, and build an awesome piece of productivity software. It's a wonderful and sophisticated task that is a capstone to all you've done so far. We can't think of a better way to top off our intense journey.

Due: Tuesday, November 19th at 11:59 p.m.[1]

The Assignment

Your mission is to build a simple spreadsheet, starting with a slightly modified version of the expression evaluator presented in Chapter 19. This assignment is designed to accomplish the following objectives:

- To more fully explore the notion of object-oriented programming. The program is broken down into classes that cooperatively interact. Almost every one of the classes we studied this quarter plays some role in the overall program architecture.
- To learn how C++ inheritance can be used for expression trees and how to implement simple recursive-descent parsing.
- To give you practice working with graphs and graph algorithms.
- To get a taste of the Model/View/Controller (MVC) structure used by many modern applications.
- To learn how to adapt existing code (in this case, the expression interpreter) to solve a different but related task. The majority of programming people do in the industry consists of modifying existing systems rather than creating them from scratch.
- To experience the joys and frustration of designing a class interface/implementation.

The task may sound a bit daunting, but never fear, there is a fair amount of infrastructure in place already. However, there is still much for you to do! Make it your personal goal to have your final assignment be one that genuinely rocks.

---

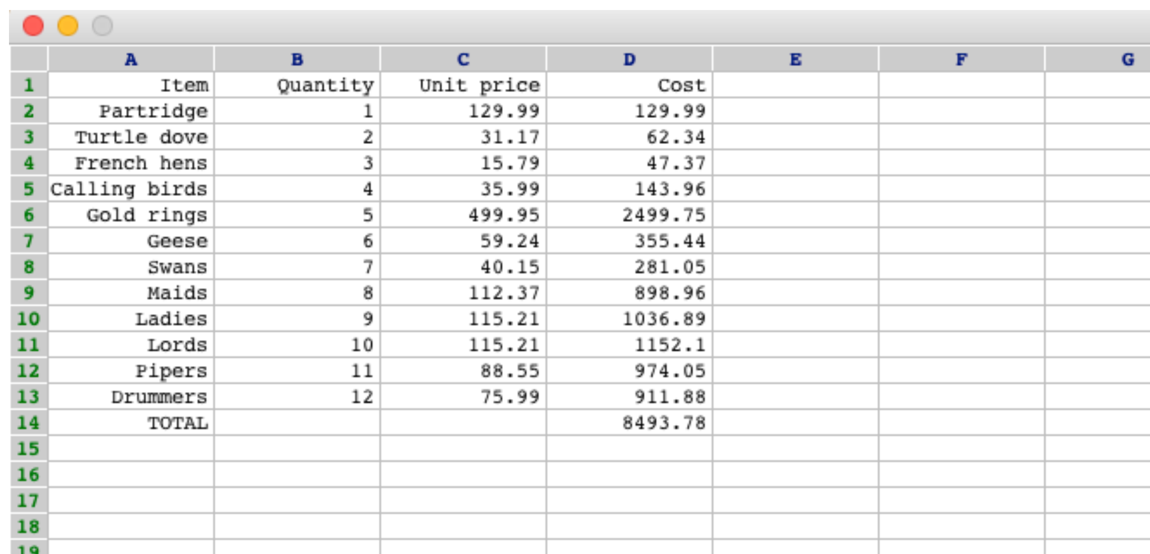
[1] Those needing an extra late day can submit as late as Thursday, November 21st at 11:59pm. Those wishing the consume a second late day can submit as late as Friday, November 22nd at 11:59pm.

A note on open-ended design

Although we give you a lot of starter code and many suggestions, this assignment is more open-ended than most and offers you the freedom to design things the way you want. There are a few isolated tasks where we mandate a particular implementation strategy, but other than that, it's up to you to make sensible decisions. Your program is expected to portray the described behavior and work similarly to the demo version, but you're being given broad authority over the choices you make to build out a working product. This kind of open-ended design can be creative and fun, but there is also the risk that you'll go astray with suboptimal choices that you later have to live with. We recommend starting the design process early and carefully thinking through alternatives and tradeoffs. We encourage you to run your design by your section leader—over email is fine—before you start coding to help you identify and correct potential problems earlier rather than later.

The Goal

One of the most important commercial programs to emerge from the personal computer revolution was the electronic spreadsheet. The original VisiCalc system was a runaway success for Apple in the early 1980s, and many more advanced products, such as Microsoft Excel and Google Sheets, have extended that basic idea so much that spreadsheet applications are now used as the basis for an astonishingly wide range of commercial use cases. At its core, a spreadsheet consists of a two-dimensional grid of cells, each indicated by a letter representing a column and a number representing a row, as illustrated in the diagram below:



	A	B	C	D	E	F	G
1	Item	Quantity	Unit price	Cost			
2	Partridge	1	129.99	129.99			
3	Turtle dove	2	31.17	62.34			
4	French hens	3	15.79	47.37			
5	Calling birds	4	35.99	143.96			
6	Gold rings	5	499.95	2499.75			
7	Geese	6	59.24	355.44			
8	Swans	7	40.15	281.05			
9	Maids	8	112.37	898.96			
10	Ladies	9	115.21	1036.89			
11	Lords	10	115.21	1152.1			
12	Pipers	11	88.55	974.05			
13	Drummers	12	75.99	911.88			
14	TOTAL			8493.78			
15							
16							
17							
18							
19							

In the spreadsheet, each cell contains a value, which can be:

- a string, such as **`"Item"`** in **`A1`** or **`"French hens"`** in **`A4`**.

- a number, such as the quantities in column **B** or the prices in column **C**. Note that these values can have decimal fractions and must therefore be represented using type **double**. (The expression hierarchy has been updated to accommodate this change.)
- a formula linking other items in the spreadsheet. Presumably, cell **D2** was set so that its value is calculated by multiplying the values in cells **B2** and **C2**. Similarly, cell **D14** is the sum of the values in cells **D2** through **D13**. Cells that reference other cells are said to have *dependencies*. If the value in **B2** or **C2** changes, **D2** will also need to be updated since it depends on those inputs.

Commands for the Spreadsheet Controller

Let's first examine the program from a user perspective and postpone our discussion of the internal machinery until we know what the program does. It opens with a new empty spreadsheet. Using a simple command-line interpreter interface reminiscent of a bad flashback to the 70s, the user can enter text commands that operate on the spreadsheet.

The command **load <filename>** reads the contents of a previously saved spreadsheet from the named file.

The command **save <filename>** writes the current contents of the spreadsheet to the named file.

The command **clear** clears the current contents of the spreadsheet.

The command **set <cellname> = <value>** sets the current contents for the given cell, replacing any previous contents for that cell. The cell name is specified by column and row such as **A3**. The value can be a string enclosed in double-quotes or a numeric expression. Below are some examples:

```
set A2 = "Beat Cal"
set B2 = 13.5
set C2 = B2 * (3 + A1)
```

If the cell name or value is invalid or malformed, an error is reported and the command discarded. Otherwise, the new contents are displayed and all cells that depend on the updated value are themselves updated.

The command **get <cellname>** prints information for a given cell which include its contents and a list of the cell's dependencies: both those cells that this one directly depends on and those cells that directly depend on it. (We'll explain more about dependencies later in this handout). For example, given the cell contents above in the **set** command, **set C2** would print:

```
C2 = (B2 * (3 + A1))
Cells that C2 directly depends on: A1 B2
Cells that directly depend on C2:
```

You are also welcome to use the **`get`** command to print any additional cell information (such as the indirect dependents) useful to your development and debugging. In fitting with our general philosophy for this assignment, your output doesn't have to match this output exactly; you just need to ensure that all our required information is present.

The command **`help`** prints a simple help message describing the available commands.

The command **`quit`** quits from the program. **`#obvi`**

Overview: Program Structure

The spreadsheet program is internally structured using the Model/View/Controller (MVC) design pattern favored by modern GUI applications. The *model* manages the data being stored. A *view* displays a visual representation of the model. The *controller* provides a user interface (be it graphical widgets or a retro command-line) that offers the user a way to make changes to the model. The controller responds to the user actions by messaging the model to update the data. When the model is changed, it notifies its views to render the new data. The benefit of MVC is that it divides the code into clean areas of responsibility, makes it possible to have multiple views/controllers on the same model, and allows you to easily try out different implementations for each component.

In the case of the spreadsheet, we supply the controller (the command-line interface) and the view (the graphical display). The model is left for you to design and implement. This class is where you will concentrate your efforts, while making compatible modifications to some of the other modules.

Here is a summary of the program structure:

`sscontroller` This module houses the **`main`** function, which is responsible for the text-based interface. It uses a **`TokenScanner`** (documentation is accessible via the CS106X web site) to break apart the command line and messages the spreadsheet model with the changes. Most of this module is written, but you'll need to extend it to support one additional command.

`SSView` This class provides a graphical spreadsheet display. We provide the complete class and you will not need to make changes to it unless you want to.

`SSModel`	This class manages the spreadsheet and cell data model. We provide a skeletal public interface and you will finish the design and provide the class implementation.
`Expression`	The Qt project uses the **`exp`** and **`parser`** modules from the Chapter 19 expression evaluator with a few adjustments. Most of this code will be used as is, but you will make additional modifications to support features required for the spreadsheet formulas.
`ssutil`	This module provides a few utility functions and the range formula functions (**`median`**, **`sum`**, **`max`**, etc.) for use in formulas. You may or may not need to make modifications to this module.

The rest of this handout focuses on the modules you'll work in and even suggests a course of action as to how you might approach the assignment. We've put the task breakdown last this time for a reason; we strongly suggest you read this handout thoroughly and get an idea of how all the pieces mesh together before you start doing any design or coding!

The `sscontroller`, `ssview`, and `ssutil` modules

We provide these three modules to you in complete or nearly complete form.

The **`sscontroller`** module contains the **`main`** program loop that interacts with the user, reading and acting on commands. It uses a **`TokenScanner`** to process the user's input and determines the appropriate action using a little command-dispatch table. Our code correctly implements the controller responsibilities, except that it is missing the **`clear`** command, which clears the current spreadsheet contents. You're to add this command to the controller.

Note that the controller is tightly coupled to the expression evaluator code from Chapter 19. One difference from the code given in the text is that the interpreter loop for the spreadsheet controller contains additional code to recover from errors. If you are entering a formula and make a syntax error, you do not want your entire spreadsheet to bomb out. Even so, it is extremely convenient in the code to call **`error`** to produce the error messages. Our implementation of the **`error`** function—which you've been seduced into believing automatically terminates program execution—actually throws an exception that's caught in the primary read-evaluate-print loop (sometimes abbreviated as the repl).

The **`ssview`** module provides the class that manages the appearance of the spreadsheet in the graphics window. It includes member functions for displaying an empty spreadsheet and displaying the contents of a cell. Your model should send messages to the view as needed to update the display when changes are made to the model. Comments in the **`ssview.h`** file describe the public features of the class.

The **ssutil** module has a few little utilities that didn't quite fit anywhere else. It defines location and range types, functions to convert a cell name to location and vice versa, and code for the range formula functions (**average**, **sum**, **max**, etc.). You are free to extend, change, and otherwise cannibalize the code here in any way you find helpful.

Hints and requirements for these modules:

Adding the **clear** command to the controller requires just a few lines of code, but you must first work through the controller code to understand how to fit the required code into the overall program architecture.

A range represents a set of cells between a start and stop cell inclusively. A range can span just one row or column or enclose a two-dimensional rectangular block of cells. Thus, ranges like **A1:A4**, **A1:A1**, **A1:D1**, and **A1:D6** are valid. One thing to note is that a valid range is required to be nonempty, which means the stop cell must be at a position that's at least equal to the row and column of the start cell.

The **range** record defined in **ssutil** stores a **location** for start and stop. An alternative definition might represent the start and stop as string cell names. In some places, you refer to a cell by its string name, other times you need its components, so both approaches will require translation. **ssutil** supplies simple conversion functions.

You can modify the supplied range formula functions to fit with your mechanism to access a range of cells (e.g. have the functions directly operate on your model) or just use the functions as given (you first extract the needed values into a **Vector**). Either approach is fine with us.

Matching a range formula function name **median** to the appropriate function to execute should be implemented using the command table approach, like that used in the controller module. It should be a simple task to add new range formula functions if you've designed it well.

Expressions and Parsing

Cell formulas can be built out of real numbers, references to other cells, parentheses, the four operators **+**, **-**, *****, and **/**, as well as built-in functions applied to cell ranges. Here are examples of some possible cell formulas:

```
5 * 1.08
A1 + A2
"Beat Cal!"
SUM(B1:B5)
(3 + A5)/average(A1:B5) + D10 - 5.5
```

With a few modifications, the **`Expression`** classes are the perfect mechanism for managing cell contents. We give you the code from Chapter 19 as your starting point, with our changes to allow floating point constants (instead of integer constants) and to add string literals enclosed in double-quotes as a new expression type. Although most of the expression code will be used as is, you do need to understand it thoroughly so you can properly leverage it for your own purposes.

The changes you are to make:

1. *Adapt expressions to work in the context of the spreadsheet model.* Whereas the ordinary expression evaluator allows arbitrary use of identifier names for variables through a variable table, variables now must be references to spreadsheet cells. Modify the parsing code so it accepts only cell names as identifiers, and update the evaluation code to retrieve the values for cells from the spreadsheet model.
2. *Add support for a new expression type of a function applied to a range.* A range function expression applies a named function to a cell range, e.g. **`sum(A1:A5)`** applies the **`sum`** function to all cells from **`A1`** to **`A5`** inclusive. The modified grammar for terms becomes:

   ```
   T  -> "string"
   T  -> number
   T  -> cellname
   T  -> function(cellname:cellname)
   T  -> (E)
   ```

   This will require adding a new **`Expression`** subclass and making changes to the parsing code. The named functions that you are required to support are listed in the **`ssutil.h`** interface file.
3. *Add support for identifying dependencies.* In order to report cell dependencies, you need to be able to find the dependent references within an expression. This is a matter of walking the expression tree and reporting the dependencies found within the sub-expressions.

Hints and requirements for expressions and parsing:

Starting from a working program that solves a different problem (in this case, the expression evaluator from the textbook) is a blessing and a curse. The **`Expression`** classes and parsing code will be a great help, but you may find yourself swimming in code at first and unsure of how to proceed. It is essential that you understand the workings of the given expression code. We recommend re-reading Chapter 19 and going over the code with a fine-toothed comb. If there's anything you don't understand, be sure to ask. You do not need to make significant modifications but figuring out how and where to make changes requires you understand the existing code base.

In the original expression evaluator, the assignment operator **=** could be part of an expression. In the spreadsheet, the **=** is a throwaway character in a **set** statement, and the expression that follows it can involve the arithmetic operators but not the **=** operator. We removed that feature from the expression modules we give you to avoid confusion.

When parsing a cell formula, the parser should reject all malformed inputs (improper cell reference, unknown range formula function, invalid range, and so on). There are a lot of cases to consider, so do be thoughtful and test carefully. Use **error** to report the problem and the exception handling in the controller will catch it and go on.

When evaluating a cell formula, a reference to an empty or string cell is assumed to have value 0. If **A1 = A2 + 5** and **A2 = "hello",** then **A1** will show the result **5**. Similarly, a function such as **sum** applied to a range of string cells would have a zero result.

Formulas should be case-insensitive: cell references **A2** and **a2** and functions **median** and **MedIAN** are the same thing. Be sure to use the **virtual** keyword to get the proper dynamic dispatch for any new member functions added to the base **Expression** class that are intended to be overridden by subclasses. Just for reference, the expression/parsing modifications involve changing/adding about 50 lines of code.

The spreadsheet model

Your main task is designing and implementing the **SSModel** class to manage the cell data. We provide a skeletal interface that lists exactly those public features needed to interact properly with our controller and view. You are to finish the design of the interface and implement the class. This is an excellent opportunity for you to think through the various options and make the decisions to suit yourself. In the real world, it is rare that code you must write comes to you fully specified, and we want you to gain some experience in the issues and tradeoffs that come up.

In general, the model is responsible for storing the contents of the cells, managing the relationships between cells, responding to requests from the controller, and notifying the view when things change. You're likely to find designing this class to be an iterative process—you sketch out the features, but as you move forward, you uncover problems or features that necessitate further refinement. Perhaps as the implementer you find some operations difficult or inefficient to support, or as the client, you encounter tasks that are awkward or even impossible. Back to the design drawing board, to make additions and adjustments as needed.

Most of the implementation strategy is left unspecified. Consider the options and make your own well-reasoned decisions. Keep in mind that you have the full collection of our classes at your disposal (sets, grids, maps, and so on) and more than one may be useful

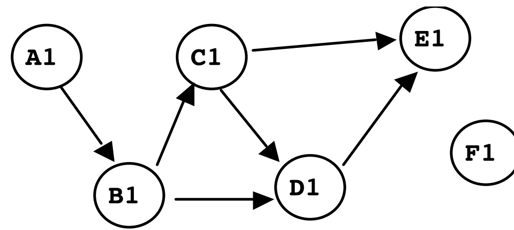
here. Here are some questions to get you started thinking about the kind of issues that you need to address:

- Should cells be created for each cell in the spreadsheet from the start or only on demand? What reasons are there to prefer one approach to the other?
- What might the model use to store cells: A grid? A set? A map? What supports easy lookup up by cell name? What about by **`location`**? What allows easy iteration/mapping over the cells? Would two ways of accessing the cells make sense or would it be overkill?
- How are the contents for each cell stored? What needs to happen when a cell's contents change?
- Should a cell cache its computed result or re-compute on the fly? If you store only the formula, each time you need the result you must re-evaluate it. Instead you could evaluate the expression once and cache the result, and use it repeatedly, only discarding and re-computing when a dependency changes. What are the tradeoffs between the two approaches?
- How might you support accessing the cells/values within a range: iteration? a function that returns a set/vector of cell names or values? What is more convenient to use/implement? Are there good reasons to support more than one technique?

One of your greatest challenges is dealing with cell formulas. The trickiness comes in the fact that changing a value in one cell may start a chain reaction of updates. A cell that has a reference to another cell in its formula is said to be *dependent* on that cell. If the value in a cell is changed, the cells that depend on it also must be updated. Dependencies can either be *direct* (where a cell has an explicit reference to another in its formula), or *indirect* (where a cell has a chain of references that eventually lead to that cell). Consider the following spreadsheet file:

```
A1 = 10
B1 = A1 * 2
C1 = B1 + 5
D1 = C1 / B1
E1 = SUM(C1:D1)
F1 = 22
```

`B1` directly depends on **`A1`**, **`C1`** directly depends on **`B1`**, **`D1`** directly depends on **`B1`** and **`C1`**, and **`E1`** directly depends on **`C1`** and **`D1`**. Both **`C1`** and **`D1`** indirectly depend on **`A1`** (through **`B1`**), and **`E1`** indirectly depends on **`A1`** and **`B1.`** One useful way to visualize the dependencies is in terms of a directed graph, as shown below:



In this graph, the arcs trace the direction of propagating/outgoing dependencies. The same arcs in reverse show the incoming dependencies. For example, when **A1** changes, it needs to propagate an update to **B1** because the formula for **B1** directly references **A1**. Indirect dependencies are those cells connected through a longer path. **D1** indirectly depends on **A1**, since it relies on **B1**, which in turn relies on **A1**, this indirect dependency is represented as a path of arcs from **A1** to **D1**.

When the value of a cell is updated, you must update all cells that depend on it, either directly or indirectly. Tracing the paths away from **A1**, you can see that changing the value in **A1** will require four other cells to be updated. Changing **F1,** on the other hand, requires no changes to any other cells, since it has no outgoing dependencies. **F1** also has no incoming dependencies, i.e., it is not affected by changes to any other cells.

Traversing the dependent cells sounds suspiciously like depth or breadth-first traversal of the graph. You can do this recursively or iteratively using a stack or a queue. A simple (and acceptable) version might update some cells multiple times because there is more than one path between them (consider how **D1** could be updated twice when traversing from **A1**). The really slick way is to do a *topological sort* to efficiently order the cell updates so that each dependent cell is updated at most once, only after its dependencies have been updated.

There's one more bit of trickiness with dependencies. What if the formula for **A1** were **sum(A1:E1)**? To calculate the value of **A1**, you need the value of **A1**, but you don't know what it is because you're still trying to calculate it! This kind of dependency is called a *circular reference*, and it's bad, bad news for spreadsheets. You should disallow all circular references. An obvious circular reference would be an attempt to set **A1 = A1**, e.g. where a cell directly references itself. The sneakier form is via an indirect reference such as assigning **A1 = E1** in the above example, which introduces a *cycle* in the graph.

Before assigning a new formula to a cell, you should traverse the graph of dependencies to ensure it will not create a cycle. Consider if the user tried to set **A1 = F1 + E1** in the above graph. The two cells directly referenced by the formula are **F1** and **E1**. We examine the incoming dependencies for **F1** and find none because it has no cells on which it depends, so this will be no problem. Next, we examine the arcs leading to **E1**, and find that it directly depends on **C1** and **D1**. So far so good, but when we continue our recursive

traversal to find the cells they depend on, we eventually run into **A1**, which is exactly what we didn't want to find. The formula is disallowed because it is circular.

Hints and requirements for ssmodel:

When setting a cell, first check for problems (invalid name, parsing issues, circular reference, etc.) and if any are found, discard the formula and leave the cell unchanged. It's best to do all the checks before making any changes, so you don't have to undo it halfway.

- The file format used for the **load** and **save** commands is a simple text file containing a list of all non-empty cells, one cell per line. Here is an excerpt from the file displayed on page 2:

```
A2 = "Partridge"
B2 = 1
C2 = 129.99
D2 = B2 * C2
```

- The ability to load and save files will be an invaluable time-saver for your testing. We provide some sample saved files in the starting project as test cases. You may assume that the contents of files being loaded are well formed, unlike the user's typo-ridden input, which must be gracefully handled at the command line.
- We suggest getting basic cell assignment, display, load/save, etc. all working without tracking dependencies first. Handle dependencies only after the underlying infrastructure is implemented and debugged.
- You're free to represent the dependencies in any way you like (using pointers, sets, vectors, etc). You'll find that you need both outgoing dependencies, i.e., those cells that must be notified when this one changes, and incoming dependencies, i.e. which cells when changed require this one to update. The incoming arcs are just the outgoing arcs reversed, but it'll be better if you store them in such a way to enable easy access in either direction. This little bit of redundancy makes some tasks simpler to manage.
- Make sure you have a basic understanding of graphs and graph traversals as detailed in Chapter 18. We recommend drawing pictures to visualize. Accidentally introducing cycles into the graph will create opportunities for infinite recursion or iteration, so be extra careful to avoid them.
- The **SSModel** class has much more room for design decisions than previous assignments. We provide some starting suggestions, but much of how you structure things is up to you. You will likely find yourself wrestling with various decisions and tradeoffs. There is no definitive "right" way, but there are better and worse choices. Part of your job is brainstorming your options, making thoughtful decisions, and documenting your reasoning. Taking the time to make good choices early in the design phase can significantly pay off during implementation. You are strongly encouraged to ask us for advice along the way— remember it is far easier to

correct a questionable decision early in the process than when the code is further along.

And don't forget these:

- Your design may have two or more classes that each depend on each other. An **`Expression`** class uses an **`SSModel`** object that in turn stores **`Expression`** objects. Think about the trouble of having **`ssmodel.h`** include **`exp.h`** while **`exp.h`** tries to include **`ssmodel.h`**! In C++, the mechanism for dealing with this is the *forward reference*. At the top of the **`ssmodel.h`** file, before you declare the **`SSModel`** class interface, you can insert a forward reference to a class it depends on, such as **`Expression`**, with this bit of syntax:

  ```
  class Expression;
  ```

  This forward reference informs the compiler that there will be a class named **`Expression`**. This allows the **`SSModel`** to happily continue on declaring data members and method parameter/return types that are of type **`Expression *`** since the compiler has been assured such a class will exist and will be around later.
- You are expected to properly dispose of any dynamically allocated memory. This is particularly tricky, because the locations of many dynamically allocated objects are aliased and held in multiple places, and it's a struggle to keep tabs on them so that everything is freed exactly one time!
- In some situations, you'll benefit from calling an **`Expression`**'s **`getType`** method to decide whether or not it's safe to downcast (e.g. explicitly cast an **`Expression *`** to, say, a **`TextStringExp *`**) so that you're able to invoke a method specific to the subclass.

Good luck with the assignment, and we hope you enjoy it and appreciate it as something you couldn't possibly have built seven weeks ago!