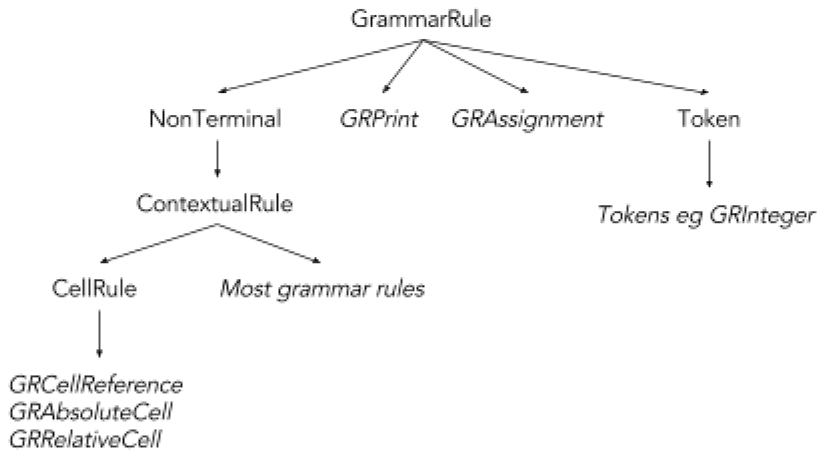
Spreadsheet Expressions

COSC 346 Assignment One | Rhianne Price 1953770

My implementation has two parts, functionality for parsing input according to the grammar, and a model for keeping track of what is in the spreadsheets' cells. For the parsing part, I have written a different class for each grammar rule, as suggested by the skeleton code. Each grammar rule is composed of more grammar rules (right hand side rules), and a specialised parsing method to change its string value and calculated value after input has been successfully parsed. I have been able to generalize some of this functionality across the grammar rules by adding some classes between the *GrammarRule* base class and the specific class for each grammar rule. The model I have used to keep track of the contents of cells is basically a dictionary of string integer pairs indexed by a cell reference, with some extra functionality to deal with stale data.

Parsing Input

I have implemented parsing for the grammar using the class structure below:



I have made use of polymorphism by overriding the parse method in each grammar rule to a more specialised parse method. Since each grammar rule still inherits from the base class *GrammarRule*, the parse method can be called on any of the grammar rules, even though they have different parse methods. All the grammar rules still have variables inherited from *GrammarRule*, so these can also be accessed regardless of which type of grammar rule is being used.

NonTerminal rules always get their string value from their right hand side rules after a successful parse. This includes most of the rules in the grammar, so was a useful generalisation to make. The parse method for all *NonTerminal* rules uses the *GrammarRule* parse method given in the skeleton code, and then sets the current rules' string value to be the concatenation of its right hand side rules' string values if the parse was successful.

Relative cell references need a containing cell to be expanded further. To begin with, I just used a single static variable to keep track of the current context cell, which was set on an assignment command and used on a print command. This approach was messy and hacky, and led to bugs when I began implementing the model. *ContextualRule*s have state that may depend on a context cell. The cell context of a grammar rule is a object property, so I decided to implement it that way. *ContextualRules* have an extra optional field that holds a *CellReference* representing the containing cell.

ContextualRules have functionality to recursively set their context and the context of their *ContextualRules* right hand side rules.. If a relative cell reference is parsed, and there is no containing cell for it to be relative to, the parse fails and returns nil. If a relative cell is parsed, and the cell it resolves to is not a legal cell (if it is out of the bounds of the spreadsheet) the parse fails and returns nil.

CellRule rules have an extra optional field for a *CellReference*. They are a subset of *ContextualRule*s, so are an extension for special cases of *ContextualRule*.

The pattern here is that after a successful parse, the grammar rule sets some of its state by reaching into its right hand side rules and doing something with their instance variables (whether it be multiplying, adding, concatenating, copying etc.). This is a form of coupling, but it is kept to a minimum because a grammar rule only ever looks into the state of its right hand rules and never any deeper. The data associated with each grammar rule is specific to that grammar rule (calculated value, string value, right hand side rules), so objects are cohesive. I considered making all grammar rule state private and using getters, but I decided that the benefits were not worth the hacks necessary to get around the inheritance problems.

Object oriented design and recursion work well for this problem because of the way the grammar is structured. Since the pattern is taking some input, operating on it, passing it up a level to the grammar rule above, each grammar rule parse method only does one computation. Each operation and the associated state is cohesively encapsulated in that grammar rule. Input to be parsed is broken down into objects, and one object in the program takes care of one object of the input. Each rule or object in the grammar becomes an object in the implementation. Since the grammar is recursive, my implementation of the parse method is also recursive. The recursion here goes down by splitting the input up into small chunks (the grammar rules) and then going back up by performing some computation and then passing the result up.

Model

To evaluate expressions that have one or more references to cells, some way to keep track of the values and expressions stored in each cell is needed. The Spreadsheet class has a dictionary of *CellContents* indexed by *CellReferences*. A *CellReference* implements the Hashable protocol so that it can be used as a key in a dictionary.

The spreadsheet class wraps this dictionary with add and get functions. Adding data to the spreadsheet is straightforward. The implementation of the get function for the spreadsheet model is model complex than the add function, because expressions on the model need to be recursively reparsed to make sure the cell values and expressions are up to date. For example,

> Z1 := 2 -> add(Z1, 2)
>
> A3 := Z1 -> add(A3, 2)
>
> Z1 := 4 -> add(Z1, 4)
>
> print_value A3 -> get(A3) -> 2 (but should be 4, so expression needs to be reparsed)

Both times data from the model are retrieved by the parser, the expression in that cell on the model is reevaluated to keep everything up to date and correct. This causes problems for circular references, which lead to infinite recursion. I have assumed that this is a corner case to ignore, so a circular reference will crash the parser.

The *Spreadsheet* class is used in my implementation as a singleton class because all instances of all grammar rules need only to add data to and get data from one spreadsheet. I have chosen this over a static class with static properties and functions so that it can be used as a library if need be.

Testing

I have used XCTest to test my parser. I wrote tests for each grammar rule before writing the class and parsing method for that rule. The tests for each grammar rule check whether the parse attempts consume the correct part of the input or not, and whether the correct state is computed and recorded. Each grammar rule was working correctly before I wrote the next one, so I was able to debug each grammar rule knowing any failed tests were due to the new grammar rule. I have also written unit tests for checking the basic elements of the spreadsheet model.

The print and assignment rules introduce interactions between the grammar parsing and adding and retrieving from the spreadsheet model, so I have tested parsing of these two rules slightly differently. For testing the assignment rule, I have tested the parser against the model to make sure parsing assignment commands resulted in adding the correct data to the model. Testing for retrieving refreshed data from the model has ended up here as well. For testing *GRPrint* I have relied on checking the printed output by hand to make sure it matches up with expectations. The expectations for all of my tests are clearly outlined in the comments.

I have written out a few sample text files to use as test input for the command line interface. This testing was important because it simulates the intended use of the parser, but I found the unit tests were more important for the development process.