CS 330 Homework

Comma-Separated Expression (CSX)

1 Overview

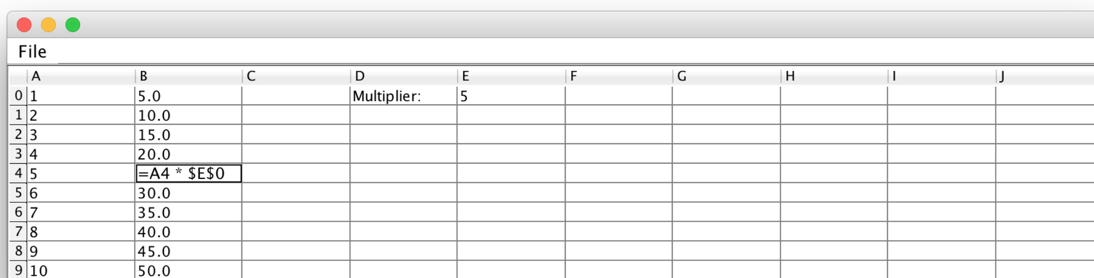
Your responsibility in this homework is to build an interpreter for text-based spreadsheets or CSX files, which are like CSV files but with formulas or expressions embedded in them. In doing so, you will gain experience writing grammars, lexing and parsing, and developing a model of an expression in an abstract syntax tree.

2 Spreadsheet

We have provided a simple GUI for editing CSX files. Class `Spreadsheet` in the JAR file at `specs/grade_csx/speccheck_csx.jar` in your repository is a `JFrame` comprised of widgets that call down to the code you write to evaluate a formula. You can run the GUI (after you've written your `Expr`, `ExprString`, and `Interpreter` classes) with the following shell command:

```
java -cp ../specs/grade_csx/antlr-4.5.2-complete.jar:speccheck_csx.jar:. csx.Spreadsheet
```

Double-click on a cell to enter text. Cells containing formula must start with `=`. For example `=5 + 6` will render as `11`, but `5 + 6` will be considered as plain text. Our language doesn't have variables, but formulas may refer to other cells via their address, as shown in the following spreadsheet:



The GUI also crudely supports filling operations. Click and drag to select several cells, and then right-click to fill formulas to other cells. In the example above, we entered `=A0 * $E$0` in cell `B0`. Then we selected `B0:B9`, right-clicked, and selected `Fill Down`.

Though we have provided a GUI, we suggest you write your own testing code—at least initially. The GUI puts a layer of separation between your tests and your code, which makes reasoning about and locating errors difficult.

3 SpreadsheetModel

Behind the spreadsheet GUI is a `SpreadsheetModel`, which stores the spreadsheet table data decoupled from its presentation. This model acts as a variable store. When a formula that you are evaluating references another cell, you will need to ask this model for the cell's value with code like the following:

```
Expr other = model.evaluate(column, row);
```

Probably you will not need to know any more about this class. However, you might find it useful to create your own model instances as you test your code. The possibly relevant public interface for `SpreadsheetModel` is as follows:

```
SpreadsheetModel(String filePath) // constructs a model from a given CSX file
SpreadsheetModel() // constructs a blank 10x10 model
void set(int column, int row, String cellText) // sets cell's source code
Expr evaluate(int column, int row) // evaluates cell's AST
String getSource(int column, int row) // gets cell's source code
Expr getTree(int column, int row) // gets cell's unevaluated AST
int getRowCount() // gets number of rows in table
int getColumnCount() // gets number of columns in table
```

4 Requirements

To receive credit for this homework, you must satisfy the requirements described in the section below. There are a number of classes you must write, but you should find that in fleshing out your abstract syntax tree hierarchy a lot of code will be reusable. The following general requirements must be met:

1. Place all files in directory `<YOUR-REPOSITORY>/csx`.
2. All code must run on `thing-0[456]`.
3. All Java classes must be placed in package `csx`.
4. Use the ANTLR 4.5 JAR file that is included in `specs/grade_csx`.

4.1 Address

Write a class `Address` that represents the address of a cell in a spreadsheet. It has the following public interface:

1. A constructor accepting four parameters: an `int` column, a `boolean` column lock, an `int` row, and a `boolean` row lock. If a lock is `true`, then the locked component of the address won't shift when translated. Translation is triggered in the GUI when the user fills a formula across a range of cells.

2. A constructor accepting a `String` parameter identifying the address by a textual description. For example: `A5` refers to column 0, row 5; `B12` refers to column 1, row 12. Assume that the column is in [A, Z] and the row is in [0, 100]. If either the row or column of this address is *locked*, then it is preceded by `$`. For example, `$A5` is column-locked, `A$5` is row-locked, and `$A$5` is both row- and column-locked.

3. Getters `getRow` and `getColumn`, which return the row and column as `int`s.

4. Method `translate`, which accept two `int` parameters: a column offset and a row offset. It yields a new `Address` that is an independent copy of this one with this twist: if this address is not column-locked, then the new address is shifted horizontally by the column offset, and if this address is not row-locked, then the new address is shifted vertically by the row offset. For example, `new Address("C$1").translate(3, 5)` → `F$1`.

4.2 CSX Grammar

Write an ANTLR grammar named `CSX`. Support the following:

1. parenthetical grouping of expressions

2. primitives: integers (positive and negative), reals (positive and negative), and addresses of the form `A5` or `$F10` or `$T$36`.

3. unary operators: arithmetic negation (not logical)

4. binary operators: exponentiation (using ^), addition, subtraction, multiplication, division, and modulus

5. collection operators: `sum`, `min`, `max`, and `mean`, which have the form `name(from:to)`, where `from` and `to` are addresses. For example, to calculate the mean of the first 5 rows and 3 columns, we write `mean(A0:C4)`.

This grammar will look similar to the ones we wrote in class. To produce the Java lexer and parser for your language in package `csx`, invoke `antlr4` in the following manner:

```
antlr4 -package csx -o csx CSX.g
```

4.3 Interpreter

Write a class `Interpreter` containing all the callbacks that ANTLR will trigger as it walks an expression's parse tree. This will look very much like the ones we wrote in class. It has at least the following public interface:

1. `static` method `parseToAST` that accepts an expression in `String` form and returns the expression as a tree of `Expr`s, described below. For example, `parseToAST("5 + 8 * 2.0")` → `new ExprAdd(new ExprInteger(5), new ExprMultiply(new ExprInteger(8), new ExprReal(2.0)))`.

4.4 Expr Hierarchy

The following requirements involve the classes in your expression hierarchy. They allow you to model the source expressions in an abstract syntax tree.

4.4.1 Expr

Write an abstract class `Expr` that serves as the superclass of all possible expressions. It has the following public interface:

1. A constructor accepting an `int` parameter for the expression's precedence level. The higher the number, the greater its precedence.

2. Abstract method `evaluate` that accepts a `SpreadsheetModel` parameter and yields an `Expr`. Subclasses will override this to produce an appropriate value given the type of expression it implements.

3. Abstract method `translate` that accepts two `int` parameters: a column offset and a row offset. It returns an `Expr`. Subclasses will override this to make a shifted copy of this expression.

4. Abstract method `toSource` that returns a "source code" representation of this expression as a `String`. Subclasses will override this method to turn parts of the abstract syntax tree back into code that can be written to a file.

5. Methods `toInteger`, `toReal`, and `toString`, which yield `int`, `double`, and `String`, respectively. Subclasses will override this method to "cast" the expression's value to a certain type. At this abstract level, we don't really know what casting means. However, instead of making this method abstract, have it throw `UnsupportedOperationException`. If a subclass chooses not to support a particular cast, we can just rely on the inherited behavior to generate a runtime error.

6. Method `getPrecedence` that returns the expression's precedence as an `int`.

4.4.2 Primitives

The following requirements involve the simplest kind of expressions we'll encounter: plain old values.

1. Write an abstract class `ExprPrimitive` that serves as the superclass of all possible expressions comprised of just an atomic value. It has the following public interface:

   (a) A constructor that sets this expression to have a precedence of 60.

   (b) Method `toSource` that returns the source code representation of this expression as its regular `String` representation, as returned by `toString`.

2. Write classes `ExprInteger`, `ExprReal`, and `ExprString` that serve as primitives in our abstract syntax tree. They have the following similar interface:

   (a) `ExprPrimitive` as a superclass.

   (b) A constructor that accepts a parameter for the corresponding Java value (`int`, `double`, or `String`).

   (c) Methods `toInteger`, `toReal`, and `toString`. Implement these methods to allow the caller to switch between types. For example, `ExprString.toString` returns the underlying `String`, while `ExprString.toReal` extracts a `double` from the `String`, and `ExprDouble.toInteger` truncates the underlying real number. To convert a number to a `String` just concatenate with the empty `String`. Reals without digits after the decimal point will render with a trailing `.0`. For example, `new ExprReal(17).toString()` → `"17.0"`.

   (d) Overridden method `evaluate`. Since primitives are already fully evaluated, just return this expression without any further computation. When someone attempts to evaluate a complicated expression tree, primitives serve as the recursion-stopping leaf nodes.

   (e) Overridden method `translate` that returns an independent copy of this expression, shifted by the specified offsets. Since shifting has no effect on primitives and primitives are immutable, you may safely return this expression directly—without first making an independent copy.

3. Write class `ExprAddress`. This class serves as a variable in our expression language. However, instead of referring to named data, this kind of expression refers to another cell in our spreadsheet. It has the following public interface:

   (a) A constructor accepting an `Address` parameter.

   (b) Overridden method `evaluate` that looks up the value identified by the address in the `SpreadsheetModel` environment passed as a parameter. The model class has a method `evaluate(int column, int row)` that returns the primitive value of this cell wrapped up in an `Expr`.

   (c) Overridden method `translate` makes an independent, shifted copy of this expression—observing any locks.

4.4.3 Unary Operators

The following requirements involve expressions of the form `OPERATOR expr`.

1. Write an abstract class `ExprUnaryOperator` that serves as the superclass of all possible operators that apply to just one subexpression. It has the following public interface:

   (a) A constructor that accepts three parameters: a subexpression, the unary operator's symbol, and its precedence as an `int`.

(b) Method `toSource` that returns the source code representation of this expression as the operator symbol followed immediately by the source representation of the subexpression. If the subexpression has lower precedence, surround it with parentheses.

2. Write class `ExprNegate`. This class is the only unary operator we will support. It has the following public interface:

   (a) `ExprUnaryOperator` as a superclass.

   (b) A constructor accepting one `Expr` parameter: an operand to negate (read: make negative). It registers this expression as having symbol `-` and precedence 40.

   (c) Overridden method `evaluate` that computes this operation by recursively evaluating the operand and negating the result. Assume all arithmetic produces a `double`—we won't overload operators as is done in C and Java.

   (d) Overridden method `translate` that returns a translated copy of this expression. Translation is a recursive process.

4.4.4 Binary Operators

The following requirements involve expressions with two operands and having the form `expr OPERATOR expr`.

1. Write an abstract class `ExprBinaryOperator` that serves as the superclass of all possible operators that apply to two subexpressions. It has the following public interface:

   (a) A constructor that accepts four parameters: a left subexpression, a right subexpression, the binary operator's symbol, and its precedence as an `int`.

   (b) Method `toSource` that returns the source code representation of this expression as the source representation of the left subexpression immediately followed by the operator symbol (sandwiched by a space character on either side) followed immediately by the source representation of the right subexpression. If a subexpression has lower precedence, surround it with parentheses.

2. Write classes `ExprAdd`, `ExprSubtract`, `ExprMultiply`, `ExprDivide`, `ExprRemainder`, and `ExprPower`. These classes all have a similar interface:

   (a) `ExprBinaryOperator` as a superclass.

   (b) A constructor accepting two `Expr` parameters: a left operand and a right operand. Addition and subtraction use symbols `+` and `-`, respectively, and have precedence 10. Multiplication, division, and remainder use symbols `*`, `/`, and `\%`, respectively, and have precedence 20. Exponentiation has a precedence of 30.

(c) Overridden method `evaluate` that computes this operation by recursively evaluating the two operands and merging the results according to the operation. Assume all arithmetic produces a `double`—we won't overload operators as is done in C and Java.

(d) Overridden method `translate` that returns a translated copy of this expression. Translation is a recursive process.

4.4.5 Functions

The following requirements involve expressions that operate on a range of cells and have the form `FUNCTION-NAME(FROM-ADDRESS:TO-ADDRESS)`.

1. Write an abstract class `ExprFunctionCall` that serves as the superclass of all possible functions that compute a value across a block of cells. It has the following public interface:

   (a) A constructor that accepts three parameters: a from `Address`, a to `Address`, and a function name as a `String`. It registers this expression with precedence 50.

   (b) Method `toSource` that returns the source code representation of this expression as the function's name, a left parenthesis, the from address, a colon, the to address, and a right parenthesis.

2. Write classes `ExprMax`, `ExprMin`, `ExprSum`, and `ExprMean`. These classes represent the operations that can be applied to collections of data. They all have a similar public interface:

   (a) A constructor accepting two `Address` parameters: the address of the top-left cell of the collection and the address of the bottom right.

   (b) Overridden method `evaluate` that traverses the cells within the given address range and computes the appropriate value (the maximum, minimum, sum, or mean). Assume all arithmetic produces a `double`—we won't overload operators as is done in C and Java. Use `SpreadsheetModel.evaluate(int column, int row)`—which returns a primitive wrapped up in an `Expr`—to look up the value of cells.

   (c) Overridden method `translate` that returns a translated copy of this expression. Translation is a recursive process.

5 Later Week

To be eligible for later-week submission, you must have all required classes and methods. They do not need to accomplish their specified tasks, but they must exist. We suggest creating all classes and methods, but having the methods do nothing or return `null` or 0.

6 Submission

To submit your work for grading:

1. Run the grading script from your homework directory using `../specs/grade`.
2. Commit and push your work to your repository.
3. Verify that your solution is on Bitbucket by viewing your repository in a web browser.

A passing grading script does not guarantee you credit. Your grade is conditioned on a few things:

- You must meet the requirements described above. The grading script checks some of them, but not all.
- You must successfully submit your code to your repository. Expect to have issues with Git.
- You must not plagiarize. Write your own code. Talk about code with your classmates. Ask questions of your instructor or TA. Do not look at others' code. Do not ask questions specific to your homework anywhere online but Piazza. Your instructor employs a vast repertoire of tools to sniff out academic dishonesty, including: drones, moles, and a piece of software called MOSS that rigorously compares your code to every other submission. You don't want to live in a world serviced by those who squeaked by through questionable means. For your future self, career, and family, do your own work.

The grading script allows you to signal your instructor when requirements are met. You only need to send an email if you qualified for later week submission and are resubmitting after the original deadline.