CS 330 Homework

Comma-Separated Expression

1 Overview

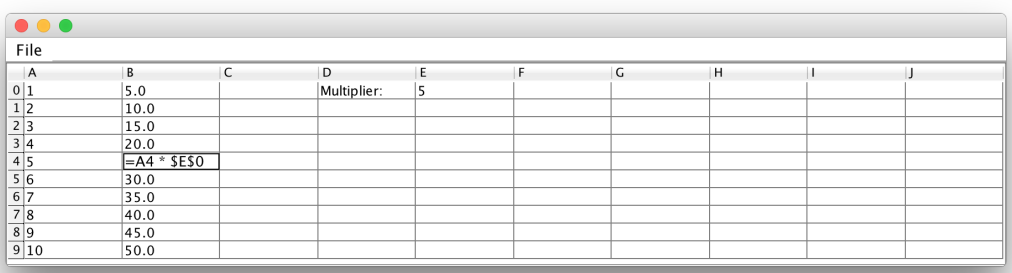
Your responsibility in this homework is to build an interpreter for text-based spreadsheets, which are essentially CSV files with formulas or expressions embedded in them. In doing so, you will gain experience writing grammars, lexing and parsing, and developing a model of an expression in an abstract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

2 SpecChecker and GUI

To help you test your code, we've provided a JAR file at `specs/grade_cse/speccheck_cse.jar` in your repository. It contains both a simple GUI that relies on your code to evaluate a spreadsheet and the unit tests used by the grader to test your work. After adding the JAR file to the classpath, you can run the GUI in the following manner:

```
EventQueue.invokeLater(new Runnable() {
  @Override
  public void run() {
    new Spreadsheet();
  }
});
```

Double-click on a cell to enter text. Cells containing expressions to evaluate must start with `=`. Our language here doesn't have variables, but formulas may refer to other cells, 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`.

3 Requirements

To receive credit for this homework, you must satisfy the requirements described in the section below. From up here, there appear to be many. However, most of the requirements involve fleshing out a hierarchy for expressions used to assemble an abstract syntax tree. A lot of code will be reusable.

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.

3.1 General

The following general requirements must be met:

1. Place all files in directory `<YOUR-REPOSITORY>/cse`.
2. All code must run on `thing-0[456]`.
3. All Java classes must be placed in package `cse`.
4. Write a class `Address` that represents the address of a cell in a spreadsheet. It has following public interface:
   (a) 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.
   (b) 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 to be *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.
   (c) Getters `getRow` and `getColumn`, which return the row and column as `int`s.
   (d) 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.

3.2 ANTLR

The following requirements involve ANTLR. Please use the ANTLR 4.5 JAR file, which is included in `specs/grade_cse`.

1. Write an ANTLR grammar named `CommaSeparatedExpression`. Support the following:

   (a) parenthetical grouping of subexpressions

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

   (c) unary operators: negation (the positive/negative kind)

   (d) binary operators: exponentiation (using $\wedge$), addition, subtraction, multiplication, division, and modulus

   (e) collection operators: `sum`, `min`, `max`, and `mean`. An expressing calculating the mean of the first five rows and 3 columns would have the following form: `mean(A0:C4)`.

   This grammar will look similar to the ones we wrote in class.

   Supporting "-" as both a unary and binary operator can be a little tricky. One has to make sure the "-" token doesn't have two ways of getting labeled by the terminal rules. Otherwise, only the first rule will be invoked and you will encounter parse errors on one of the two expressions involving the negative sign. To avoid complicating our grammar too much, I suggest a very simple and very hackish workaround: force the subtraction operator actually be two characters, a negative sign and a trailing space (`-␣`). This forces us to put whitespace around our operators, which you can assume all the tests do.

   To produce the Java lexer and parser for your language in package `cse`, invoke `antlr4` in the following manner:

   ```
   antlr4 -package cse -o cse CommaSeparatedExpression.g
   ```

2. 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:

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

3.3 Expr

The following requirements deal with `Expr`, the superclass of all expression types used to model a formula in a spreadsheet.

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

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

(b) 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.

(c) 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.

(d) 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.

(e) 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 these methods abstract, have them throw an `UnsupportedOperationException`. If a subclass chooses not to support a particular cast, we'll get a runtime error.

(f) Method `getPrecedence` that returns the expression's precedence as an `int`.

3.3.1 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.

(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.

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 and shifted copy of this expression—as the locks allow.

3.3.2 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 an independent copy of this expression. (This means the operand expression given to the copy must also be independent.)

3.3.3 Binary Operators

The following requirements involve expressions of 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 represent binary operations (read: having two operands) and 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 an independent copy of this expression. (This means the operand expressions given to the copy must also be independent.)

3.3.4 Functions

The following requirements involve expressions of the form `FUNCTION-NAME(ADDRESS:ADDRESS)`.

1. Write an abstract class `ExprFunctionCall` 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 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 an independent copy of this expression.

4 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 are resubmitting.