Java Spreadsheet

Parsing and Evaluation

COMP40009 – Computing Practical 1

13th – 17th February 2023

1 Aims

- To learn how to represent abstract syntax trees in an object oriented language.
- To practice using the Java Collections library.
- To design an interactive command-line tool, using Java's standard input and output.

2 The Problem

Spreadsheets are powerful tools for entering, modelling and viewing numerical data. At their core is the ability to represent and evaluate symbolic expressions which are stored in, and which can refer to other, *cells*. This means a change in the computed value of a cell can require other cells to recompute their values.

This lab is divided into two parts. During the first week (this exercise), you will implement parsing and evaluation of cell expressions. The second week will require that you integrate your parser and evaluator in a spreadsheet GUI, and determine what cell values must be recomputed when the user edits a cell.

2.1 Spreadsheets

Spreadsheet applications typically involve a two-dimensional grid of cells. Cells are referred to by a combination of their column and row names. Columns names are named in sequence `"a"`, `"b"`, ..., `"z"`, `"aa"`, `"ab"`, etc. Rows are named incrementally starting from 1. For example, `"a2"` names the second cell down in the left-most column and `"ba5000"` names the cell in row 5000, column 53.

2.1.1 Expressions

Each cell contains an *expression*. A user can edit the expression in any cell, and when editing is finished, the expression is evaluated automatically to yield a value that is displayed in the cell. Expressions can take several forms; for the purposes of this lab we consider the following:

Numbers – if the expression is a (positive) number (e.g. `"2"`, or `"2.8"`), its value is the corresponding double-precision floating point number.

Cell references – if the expression references a cell location, its value is the value of that cell.

Binary operator applications – if the expression is the application of a binary operator to two subexpressions then the value is the result of applying the operator to the values of the two subexpressions, computed recursively. In this exercise a binary operator can be one of addition (+), subtraction (−), multiplication (∗), division (/) or exponentiation (^).

Figure 1 shows an example of spreadsheet cells and the values computed from the given set of expressions.

```
a1: b1 + 1
b1: a2 + b2
a2: 25
b2: a2 + 7
```

	a	b
1	58.0	57.0
2	25.0	32.0

Figure 1: A simple spreadsheet

3 What To Do

For this exercise, we will only focus on parsing and evaluating cell expressions, without worrying about cell dependencies and spreadsheet management.

3.1 Skeleton Project

Clone your exercise repository (remembering to replace *login* with your login) with:

```
> git clone
  https://gitlab.doc.ic.ac.uk/lab2223_spring/javaspreadsheet_login.git
```

You will notice that the skeleton files have been zipped and encrypted with a password. In order to extract them, you will need to use an application that supports 7z files[1]. On the lab machines, this has already been installed for you, so you can extract the archive by typing:

```
7z x skel.7z -p05FB49D3
```

(*n.b., 05FB49D3 is the password*). Make sure that you extract the contents of the archive directly inside your repo, i.e. do not create an extra subfolder named "skel", because that would break the LabTS autotesting process. Once this is done, you can then safely delete the 7z archive. Remember to commit and push all the extracted files back into the repo.
Once the project has been imported into an IDE you should find a `src` directory containing multiple packages:

- `spreadsheet` this is where you will place all of your code. A few skeleton classes are provided, but you will need to define some more.
- `common.lexer` provides a lexer implementation and a `Token` class. You should use this package when implementing the parser, in Section 3.3.
- `common.api` contains a few classes and interfaces used by different packages to interact together.

3.2 Representing the AST

Rather than manipulate the string representation of expressions, it is easier to represent them as *abstract syntax trees* (AST). An AST is a recursive data structure and will be implemented here as a Java object capable of representing numbers, cell references and binary operator applications, as described above. Figure 2 gives an example of the AST representation of `1.0 + a2 * 3.0`. As you can see, the structure of the tree reflects the precedence of the operators `+` and `*`.

[1] `https://www.7-zip.org/download.html`

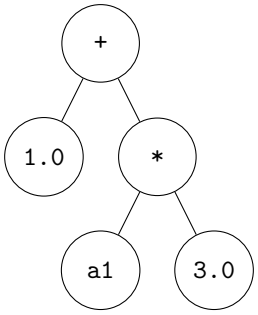

Figure 2: Abstract syntax tree for `1.0 + a1 * 3.0`

For each kind of expression described in Section 2.1.1 you should create a new class in the `spreadsheet` package. Each of these should implement the existing `api.Expression` interface. For the moment you may leave the methods required by the interface empty (or make them return an arbitrary value). We will cover them in Section 3.4.

Binary operator expressions are composed of sub-expressions, and should therefore have fields of type `Expression`. On the other hand, numbers and cell references, are *atoms*. They are made up of a plain value, with no sub-expression. You can use the `common.api.CellLocation` class to store a cell location.

You must also override the `toString` method in each class to return a string representation of the corresponding AST node. This representation should look similar to the expression language, but with additional parentheses to reflect the precedence and associativity of the various operators: a single set of parentheses should be placed around all binary operator applications, but not around literals and cell references. The `toString` method will be useful both for debugging, allowing you to print your parser's output, and for the test suite, which uses `toString`'s output to check the behaviour of your parser.

The test suite is lenient with regards to spacing and precision of floating point values, so adding or removing spaces or extraneous zeroes will not affect the test result, e.g. `01.000 + 2.0` and `1.0+2.0` will be treated as the same expression. However it is strict with regards to the number and placement of parentheses in the output, e.g. `(1.0+2.0)` and `((1.0+2.0))` will be treated as different expressions. Figure 3 contains a few examples of input expressions and their corresponding `toString` output.

Input expression	Expected `toString` output
`1`	`1.0`
`1 + 2`	`(1.0 + 2.0)`
`1 + 2 + 3`	`((1.0 + 2.0) + 3.0)`
`1 + a1 * 3`	`(1.0 + (a1 * 3.0))`
`c4 + 2`	`(c4 + 2)`

Figure 3: Examples of expressions and `toString` output.

3.3 Parsing

Once you have defined your AST nodes, you are ready to implement the expression parser. The skeleton project contains a class `Parser`, with a static `parse` method that you need to implement. The method takes a single `String` argument and returns the corresponding `Expression`.

3.3.1 Lexical analysis

In order to parse a string into an AST it is customary to break up the input into a sequence of *tokens*, through the use of a *lexer*. Tokens represent the indivisible components of expressions; here the possible tokens are ("literal") constants, cell identifiers and binary operator symbols. For example the string `"1.5 + 2.0 * a4"` would be broken up into the tokens `1.5`, `+`, `2.0`, `*` and `a4`. Writing a parser in terms of tokens is simpler than in terms of characters, as the removal of whitespace and the detection of symbol boundaries will have already been resolved. We have provided you with complete `Lexer` and `Token` classes already.

Operator	Precedence	Associativity
Addition (+)	1	Left
Subtraction (-)	1	Left
Multiplication (*)	2	Left
Division (/)	2	Left
Exponentiation (∧)	3	Right

Figure 4: Precedence and associativity of operators

The `Lexer` class should be constructed with a string – the textual representation of an expression. You may then call the `nextToken()` method in a loop to get all tokens in the string. The method returns `null` when there are no more tokens left. For example, the following piece of code will tokenize the `"1.5 + 2.0 * a4"` and print each token to the standard output.

```
Lexer lexer = new Lexer("1.5 + 2.0 * a4");
while (true) {
  Token token = lexer.nextToken();
  if (token == null)
    break;
  System.out.println(token);
}
```

Each token has a kind and possibly a value. The kind is used to categorize the token and is an enum with values such as `PLUS`, `MINUS`, `NUMBER` or `CELL_LOCATION`. If the token has kind `NUMBER` or `CELL_LOCATION`, then its contents can be found respectively in fields `numberValue` or `cellLocationValue`.

3.3.2 Shunting yard algorithm

Given the sequence of tokens, you are now in a position to parse these into an AST. We recommend doing so using Dijkstra's *shunting yard algorithm*. The algorithm works by maintaining two stacks, one for pending operands and one for pending binary operators. You should make use of the Collections Library's `Stack` class to represent those. The first stack contains fully formed AST nodes and should have type `Stack<Expression>`. The second stack contains only information about which binary operator is pending. You are free to represent this as you see fit.

Tokens are processed one at a time and handled as follows:

- If the current token is a `NUMBER` or `CELL_LOCATION`, create an equivalent AST node and push it onto the operand stack.
- If the current token is a binary operator (e.g.. `PLUS`, `MINUS`, `CARET`, ...) and the top element of the operator stack has a strictly higher precedence than the current token, form a new binary application node by combining the top of the operator stack with the top two elements of the operand stack. This also applies if the top of the stack has the same precedence as the current operator *and* is left-associative. You may need to repeat this operation multiple times with the same current token.
- Otherwise the current token should be added to the operator stack, as we are not yet in a position to determine its operands.

Not all kinds of tokens returned by the `Lexer` will be supported by your parser. For example no valid expression makes use of the token `LANGLE`. If you encounter such a token, you may throw an exception, such as `InvalidSyntaxException`.

The precedence and associativity of operators follow the conventional rules, as shown in Figure 4. The precedence values have little meaning in isolation; they are are only used to define a ordering between the operators.

When you've processed all the tokens, you should process any remaining operators left on the operand stack: repeatedly pop one operator and two operands from the stacks, combine those into an AST node and push the result to the operand stack, until no operators left. The expression at the top of the operand stack is the result of the parser.

Step	Tokens	Operator stack	Operand stack
1	`1 + 2 * 3 ^ 4 + 5`		
2	`+ 2 * 3 ^ 4 + 5`		1
3	`2 * 3 ^ 4 + 5`	+	1
4	`* 3 ^ 4 + 5`	+	2 1
5	`3 ^ 4 + 5`	* +	2 1
6	`^ 4 + 5`	* +	3 2 1
7	`4 + 5`	∧ * +	3 2 1
8	`+ 5`	∧ * +	4 3 2 1
9	`+ 5`	* +	(3^4) 2 1
10	`+ 5`	+	(2 * (3^4)) 1
11	`+ 5`		(1 + (2 * (3^4)))
12	`5`	+	(1 + (2 * (3^4)))
13		+	5 (1 + (2 * (3^4)))
14			((1 + (2 * (3^4))) + 5)

Figure 5: Parsing the expression "`1 + 2 * 3 ^ 4 + 5`"

Figure 5 shows each step of the parser as it processes the string "`1 + 2 * 3 ^ 4 + 5`". At each step, the token, operator or operands used are highlighted.

- At steps 1, 3, 5, 7 and 12, the current token is a number, which is pushed to the top of the operand stack.
- At steps 2, 4, 6 and 11, the operator stack is either empty or its top element has a lower precedence than the current token. That token is therefore pushed onto the operand stack.
- At steps 8, 9 and 10, the top element of the operand stack has a higher precedence than the current token. The top operator and the top two operands are removed to form a new operand.
- Finally, at step 12, there are no tokens left so the remaining operators are removed and assembled to form an operand.

For the purpose of this exercise, you may assume that input expression are always well-formed. You are for example not required to handle an expression such as `"1 + + 2"` gracefully.

3.4 Implementing `Expression`'s methods

In Section 3.2 you will have created a number of classes that implement the `Expression` interface. You should now implement its methods on each class.

`evaluate` computes the `double` value corresponding to the expression. The method receives one argument, an `EvaluationContext`, which is used to resolve cell references and find their values. Most operators can be implemented by using the appropriate Java equivalent operator. The only exception is exponentiation, for which the `Math.pow` method should be used.

`findCellReferences` is used to find all cells mentioned by this expression. This will become useful in next week's exercise. The method should insert referenced cells into the provided `Set<CellLocation>`. For example expression `1 + a1 * 2 + b5` references the cells `a1` and `b5`.

3.5 Maintaining the spreadsheet state

You must implement all the methods in `spreadsheet.Spreadsheet`:

- Start by implementing the `getCellValue` method. For this, the `Spreadsheet` class will need to maintain the latest value of each cell, using a `Map<CellLocation, Double>` for example. If the cell has not been defined yet (i.e. it is absent in the map), it can be treated as having value zero.
- Next implement the `evaluateExpression` method. This should parse the input string and evaluate the expression. Since `Spreadsheet` implements `EvaluationContext`, it may be passed as an argument to `Expression`'s `evaluate` method.
- Finally implement the `setCellExpression` method. You should evaluate the expression and update the map of cell values.

3.6 Command-line interface

Finally, you should implement the `spreadsheet.Main` class to provide an interactive interface to the `Spreadsheet` class. Your program should print a prompt (`>`), wait for the user to enter a command, and print the output.

Here is an example of a session made up of multiple commands. Lines that start with the prompt are the ones entered by the user, whereas the others are printed by the program.

```
> 1 + 1
2.0
> a1 = 5.0
> a2 = a1 + 10
> a2
15.0
> a1 * a2
75.0
```

Each command is made up of an expression which needs to be parsed, evaluated and the result printed. If the command starts with a cell location and an equals sign, the result value should be stored as the new value of this cell. Otherwise, if the command is just an expression, the result value need to be shown to the user.

In order to read from the program's input, you can wrap `System.in` in a `BufferedReader` and use its `readLine` method. It will return null when the input has been read entirely. For instance, the following piece of code reads and prints back each line entered by the user:

```
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
while (true) {
  String line = in.readLine();
  if (line == null) {
    break;
  }
  System.out.println(line);
}
```

Hint: You might find is useful to use Java's `line.split` method to process assignment commands of the form `id = expression`, e.g.

```
String[] parts = line.split("=", 2);
```

4 Suggested Extension

- Extend your parser to handle parentheses within expressions. A neat way to do this is to treat them as operators: if you get their precedences and associativities right then you should need only minimal changes to your parser to accommodate them.
- Generate useful error messages on syntax errors.
- Extend the expression language to support more operators (eg. comparison operators).

Submission

As usual, use `git add`, `git commit` and `git push` to send your created **.java** files to the gitlab server.

Then, log into the gitlab server `https://gitlab.doc.ic.ac.uk`, and click through to your `javaspreadsheet_<login>` repository. If you view the `Commits` tab, you will see a list of the different versions of your work that you have pushed. Select the right version of your code and ensure it is working before submitting to CATe via LabTS.

Assessment

```
F - E: Very little to no attempt made.
       Submissions that fail to compile cannot score above an E.

D - C: Implementations of most functions attempted;
       solutions may not be correct, or may not have a good style.

B:     Implementations of all functions attempted, and solutions
       are mostly correct. Code style is generally good.

A:     There are no obvious deficiencies in the solution or
       the student's coding style. In addition, there is
       evidence of productive testing.

A*:    As for an A -- plus the student has done additional work
       beyond the basic spec, e.g. by considering (and clearly
       commenting) interesting variations or extensions to the
       given functions; e.g. based on their own research.
```