COSC346 Assignment 1: spreadsheet expressions

Submission information

Due date: Friday, 4th September 2017, at 5 PM.

Weight: This assignment is worth 20% of the mark for the paper.

Marking: You are permitted to do this assignment individually or in pairs.

What to submit:

- A complete Xcode project including the source code and all the resources that you use.
- A report in PDF format. (See instructions below.)

How to submit: Collect everything that you need to submit into a directory, and then within MacOS on a Lab machine, run:
`/home/cshome/coursework/bin/submit346` *directory_name*

A submission from one member of a pair is sufficient if you choose to work in a pair. Late submissions will incur a 10% penalty per working day, rolling over at 5 PM. Submissions over five days overdue will not be accepted.

Be careful to ensure that your submission only relies on files that you have actually submitted.

Learning objectives

The aims of the assignment include the following learning objectives:

- To demonstrate programming skills using the Swift programming language, and some of the Foundation classes.
- To employ in practice fundamental concepts in object-oriented programming such as polymorphism, inheritance, design patterns, coupling and cohesion.
- To reflect on the strengths and weaknesses of Swift and object-oriented programming techniques.
- To deliver software that is well tested and documented.

Problem Description

In this assignment you will design and implement and document a tool for reading and interpreting a basic form of spreadsheet. For the core of the assignment, you must depend on nothing more than the Foundation framework. If you choose to extend your code and use further frameworks or libraries, you must also still provide a version of your code that relies only on the Foundation framework. You are also required to implement and document a testing framework for your assignment code.

You must read in an input file that specifies the contents of given 'spreadsheet' cells, and issues commands to display a textual representation of the

values of their expressions, or the contents of the cells. The file may update cells that had already been previously set, so the directives in the file must be processed in order.

For example, given the following input file:

```
A1 := 1
A2 := "covfefe"
print_expr A1
print_value A1
print_expr A2
print_value A2
A2 := 2
A3 := A1+A2
print_expr A3
print_value A3
A1 := r1c0 + 1
print_value A3
```

Your program should produce the following output:

```
Expression in cell A1 is 1
Value of cell A1 is 1
Expression in cell A2 is "covfefe"
Value of cell A2 is "covfefe"
Expression in cell A3 is A1+A2
Value of cell A3 is 3
Value of cell A3 is 5
```

Note that your program's output for `print_value` should match this form as closely as possible, since some of the functionality of your code will be tested by automated marking scripts.

To draw an analogy to a spreadsheet application, `print_value` should output the value that would be shown in a spreadsheet cell, even if that cell contains a formula. In contrast, `print_expr` should output what the spreadsheet application would display when editing the formula within a cell—you can use whatever syntax best works for you, but explain it in your report.

Your implementation of `print_expr` should expand expressions at least one step, so given an expression `A3`, you should at least describe the expression contained in that cell. You may expand the expression as far as you can, if that is easier for you. Note that some expressions do not make sense without the context of a containing cell, so in the above output the last expression for `A1` contains `r1c0`, which refers to the cell in the row below `A2`. However the expression `r1c0` by itself with no context cannot be expanded further. Ideally your code should handle this situation in some sensible way, but the test cases for marking are likely to focus on more typical use cases.

Spreadsheet expression grammar

The complete grammar for the file format is shown in Figure 1, and the syntax used in that figure is described below. You should use a recursive descent parser to satisfy the core requirements of the assignment. May read the entire input

file into a Swift String if this makes your processing easier—post-submission testing will not employ gigantic files.

You do not need a comprehensive understanding of recursive descent parsing to complete this assignment, since alongside this assignment specification on the COSC346 website, you should find some skeleton code that you should use as the starting point of your parser. (Note that recursive descent parsers are unlikely to be used in high-performance production code, but they provide a direct, object-oriented route from the grammar to the code that parses it.)

Spreadsheet	→	Assignment | Print | ϵ
Assignment	→	AbsoluteCell `:=` Expression Spreadsheet
Print	→	`print_value` Expression Spreadsheet | `print_expr` Expression Spreadsheet
Expression	→	ProductTerm ExpressionTail | QuotedString
ExpressionTail	→	`+` ProductTerm ExpressionTail | ϵ
QuotedString	→	`"` StringNoQuote `"`
ProductTerm	→	Value ProductTermTail
ProductTermTail	→	`*` Value ProductTermTail | ϵ
Value	→	CellReference | Integer
CellReference	→	AbsoluteCell | RelativeCell
AbsoluteCell	→	ColumnLabel RowNumber
ColumnLabel	→	UpperAlphaString
UpperAlphaString	→	*A string only containing uppercase letters.*
RowNumber	→	PositiveInteger
RelativeCell	→	`r` Integer `c` Integer
StringNoQuote	→	*A simple string of the usual type you might expected, but with the restriction that it cannot contain quote characters, in order to keep the grammar simple.*
PositiveInteger	→	*Integers greater than zero.*
Integer	→	*Positive and negative integers.*

Figure 1: Simple spreadsheet grammar

The grammar in figure 1 uses different typefaces, symbols and coloured backgrounds to highlight different parts of the syntax. We will step through an example parsing run below, too.

Terminal symbols A term of the form `print_value` is a "terminal symbol". It is content that you expect to find literally in the input string.

Non-terminal symbols A term of the form Value is a "non-terminal symbol". It describes some part of what you have parsed from the input string. When parsed successfully, objects that represent instances of non-terminal symbols are likely to contain properties that store the result of the parsing action. For example, the instance of an Expression that ends up successfully parsing the string `1+3*2` is likely to record that the computed value is the integer 7.

Description Text that is formatted *in italics* is a description designed to be read and interpreted by you. For example, while we could define non-terminal Integer in terms of terminal symbols `0`, ..., `9` and `-`, we are

assuming that you can determine how to implement the parsing of Integers without a formal grammar.

Grammar rule Each rule in the grammar is represented as a left-hand-side non-terminal, followed by the symbol → followed by a set of right-hand-side choices separated by the pipe symbol (|). This indicates that for parsing a particular input, the given left-hand-side non-terminal will take on the form of one of the right-hand-side choices, if it is able to parse the input. Each such right-hand-side choice is a sequence of terminal and/or non-terminal symbols.

Empty terminal, ϵ The Greek letter epsilon (ϵ) is used to represent nothing. That probably sounds rather confusing, but it is necessary: if you look at Assignment and Print, they both have another Spreadsheet term as their suffix, so if ϵ was not in the grammar, there would be no way to have Spreadsheet definitions ever finish. (Likewise for other places ϵ is used).

Some further notes:

- Integers can be negative, but since there is no subtraction or negation operator in the Expressions within this grammar, there should be no ambiguity caused by negative integers.
- Basic familiarity with spreadsheets is assumed, and as is typical, the top-left cell is A1; the cell below A1 is A2; the cell to the right of A1 is B1. To the right of column Z is column AA, then AB, *etc.*, until BA, and so on.
- Note that handling of whitespace (*e.g.*, spaces, tabs, newlines) is not covered explicitly in the grammar description. For this assignment, in most contexts you can ignore whitespace, and the skeleton code does this. In some contexts you will need to capture whitespace, though, such as spaces within a StringNoQuote.
- Feel free to focus on handling ASCII strings. We are not going to test your code on esoteric Unicode inputs.
- You can assume that cells that have not been explicitly initialised contain the Integer 0.

Recommendations

Start simple; develop incrementally

You should probably aim to incrementally develop your solution. A good strategy would be to cut the grammar down to its absolute basics at first. Perhaps something like—

Spreadsheet → Expression | ϵ
Expression → Integer ExpressionTail
ExpressionTail → + Integer

Indeed an implementation of this grammar is provided in the skeleton code. This grammar would parse the input 1+2, as follows. We can assume that we will begin parsing by attempting to match the top-left-most non-terminal, in this case a Spreadsheet. So we look at the first rule, and try to expand this

Spreadsheet into the first right-hand-side choice, which is an Expression. We then expand the rule for Expression to seek out a sequence of an Integer followed by an ExpressionTail. The Integer will successfully match the (terminal) 1. Then the ExpressionTail will match the + followed by the second Integer which will match the 2. We have now expanded all the non-terminal symbols, so have completed. It would also be valid to pass in the empty input, for which the attempt to consume an Expression from the input will fail, and we can try the second option for Spreadsheet, which is matching no remaining input (ϵ).

Test and develop your code in step

You are strongly recommended to build your test cases as you write your code for the assignment. While testing code as you build it does slow down the rate at which code appears in your assignment, for many it will also slow down the rate at which *buggy code* appears in your assignment.

Keep in mind that you need to submit your testing code as well as your implementation of the spreadsheet processing code.

Report

You must provide a report in PDF format. It is an important part of the assignment. Your report should indicate:

- the way in which object-oriented concepts were used in your design and implementation;
- how you tested your code;
- if you completed the assignment in a pair, you must explain the role taken by each member of your pair; and
- if you implemented any extensions, how many bonus marks (up to 3) you believe you should be awarded, and why. Note that the bonus marks can only be used to reach the maximum mark of 20.

The report usually should not need to be longer than one page. You will lose marks if the report has obvious typos, spelling or grammatical errors.

Rough guide to marks

The assignment is worth 20% of your COSC346 mark and will be marked out of 20. There is not a strict marking scheme, as having one tends to disadvantage students, but the marking principles will include the following considerations:

- Elegant code will receive higher marks than ugly code (*i.e.*, try to avoid ugly hacks, and feel free to fix hacks that you find in the skeleton code).
- Being unable to implement the whole spreadsheet grammar does not mean that you will fail the assignment, but you will need to clearly indicate what is and is not working, and why.
- Code that makes good use of object-oriented principles will receive higher marks than code that does not.

- You may improve on the skeleton code provided to you, but do not forget to document and justify what you did in the code and in the report. It is certainly not a requirement to redesign the skeleton code, however you may well find that it is not particularly idiomatic Swift, and you can improve upon it.
- Uncommented code will receive a maximum mark of 15 for the assignment.
- Assignments that do not provide a testing framework will receive a maximum mark of 12. Your `main.swift` should use your testing code usefully and comprehensively. It should also be possible to use your code as a library that does not run tests, *i.e.*, your testing should be cleanly separated from your implementation.
- An assignments that is submitted without a report will receive a maximum mark of 14.
- There are a maximum of 3 bonus marks available. You should make a case for how many bonus marks you deserve in your report.