COMP 250 Assignment 2

Instructor: Prof. Michael Langer
Posted: Thursday. Oct. 6, 2016,
Due: Wednesday Oct 19, 2016 at 23:59 (before midnight)

The same general instructions apply as in Assignment 1, in particular, the late penalties.

The TA's handling this assignment are as follows. Contact them by @mail.mcgill.ca.

- David Bourque (david.bourque)
- James Bodzay (james.bodzay)
- Florence Clerc (florence.clerc)
- Antoine Soulé (antoine.soule)
- Howard Huang (howard.huang)

David Bourque will be handling questions on the discussion board.

Question 1 (60 points)

Introduction

In this question, you will use a stack to implement a parser for a simple programming-like language. Much like the Java compiler tells you if there are syntactic errors in your code, your parser will check whether a given statement's syntax is correct or not with respect to the rules of this simple language.

One can define the syntax of a language using substitution rules. For example, a substitution rule like this:

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

means that whenever the term ***digit*** occurs in a rule, it can be replaced by the character '0', or the character '1', or '2', and so on.

Rules can be recursive: for example, using the ***digit*** rule, we can define a recursive rule to describe what a non-negative ***integer*** should be:

integer → ***digit*** | ***digit integer***

This means that an ***integer*** is a single ***digit*** (base case), or a single ***digit*** preceding an ***integer*** (recursive case). Here we allow an ***integer*** to start with 0, for simplicity.

We can define a ***variable*** in a similarly recursive way, namely as a sequence of ***letter*** characters.

letter → *a* | *b* | ::: | *z* | *A* | *B* | ::: | *Z*

variable → ***letter*** | ***letter variable***

We next use ***integer'***s and ***variable'***s to define possible ***statement'***s in a simple programming-like language, which is generated by the following substitution rules:

assignment → ***variable*** = ***integer***

bool → *true* | *false*

statement → ***assignment*** | *if* ***bool*** *then* ***statement*** *else* **statement** *end*

When you look at the starter code, you will see that there are two levels of string parsing that are done. The first level splits the input string into "tokens". The second level parses the tokens and checks if a sequence of tokens defines a valid statement. This is the part that you will implement.

There are a few points to note about the token parsing level that is given to you. The token parsing uses "regular expressions" rather than the recursive definition above. We chose to use regular expressions to split tokens just because it is easy to do so in Java. The splits into tokens are defined by the space characters. In particular, we require each ***assignment*** to be a single token, and so an ***assignment*** cannot contain any space characters. For example, "foo=23" is a valid ***assignment*** whereas "foo = 23" which contains spaces is not a valid ***assignment***. The splitter which generates the sequence of tokens from the input string would treat "foo = 23" (with spaces) as three separate tokens rather than as one ***assignment*** token. Another point about token parsing is that the keywords *if, then, else, end, true, false* are special tokens in the language. The token parser ensures that these keywords cannot be variables within an assignment. For example, "if=3" is not a valid assignment according to the token parser (see ***isAssignment()*** method in **LanguageTester**).

For any valid ***statement*** in the language, one ***assignment*** would be executed. For the case of an if-then-else statement, the one that is executed depends on the ***bool*** conditions. Your task, however, is not to decide which ***assignment*** is executed. Rather, your task is to decide whether a given input string is a valid statement or not, according to the above rules.

For example, here are some examples of valid ***statement'***s:

- "foo=23" (with no spaces, see above)
- "statement=23"
- *"if true then* a=2 *else* b=31 *end"*
- *"if false then if true then* c=5 *else* d=5 *end else* b=31 *end"*

and here are some examples of invalid ***statement'***s:

- "if=3" (invalid, keyword cannot be a variable)
- "foo = 23" (with spaces)
- "foo"
- "foo=bar"
- *"false"*
- *"if true then* a=2 *else* b=31" (missing *end*)
- *"if* a=2 *then* a=2 *else* b=31 *end" (invalid* **bool***)*

- *"if true* a=2 *else* b=31 *end"* (missing *then*)

There are four Java classes given in the starter code:

- **StringSplitter**: This class implements a "lexical analyzer", namely a **StringSplitter** object that reads a string and splits it into tokens. It is fully implemented for you.

- **StringStack**: Implementation of the Stack ADT for non-empty strings.

- **LanguageParser:** A partial implementation of the (token) parser. You need to implement the **isStatement**( ) method in this class.

- **Tester:** Used to test the correctness of your **isStatement**() method. The class has examples of how to use the **StringSplitter** class. Your **LanguageParser** class will be tested with a more extensive set of examples than what is given to you here.

  As in Assignment 1, we strongly encourage you to write and share your own tester classes.

Your task

Implement the **isStatement()** method of **LanguageParser** which processes an input string. It returns *true* if the input string defines a valid ***statement***, and it returns *false* otherwise.

Your solution must be non-recursive, that is, your **isStatement()** method cannot call itself.

Your solution must use a **StringStack**. You may use only one **StringStack** object, and your method must only do one pass over the tokens, i.e. each token will be read from the **StringSplitter** object exactly once.

To make the task simpler, you *are* allowed to push any non-empty strings you want on the stack. i.e. You are not limited to pushing only tokens from the language.

Submit a zipped folder **A2Q1submit.zip** *file* to the A2Q1 mycourses assignment folder. The folder should contain your class **LanguageParser.java**

For your interest:

The simple language defined above is an example of what is called a *context-free grammar*. The subject is very important in computer science. You will study this subject if you take COMP 330.

Also, the provided code uses Java *regular expressions* in a few places. Java regular expressions are based on a more general theory of *regular expressions* which you will also learn about in COMP 330. Regular expressions are commonly used in solving problems which involves text processing and you are encouraged to learn about them.

Question 2 (40 points)

a) (10 points)

Solve the recurrence

$$t(n) = t\left(\frac{n}{2}\right) + log_2 n \text{ , such that } t(1) = 0.$$

Assume in your solution that n is a power of 2, that is, $n = 2^m$ so $m = log_2 n$.

b) (10 points)

Verify that your answer in (a) is correct using a proof by mathematical induction, namely perform induction on the variable m.

c) (10 points)

If two positive integers n_1 and n_2 have N_1 and N_2 decimal digits, respectively, then computing their product $n_1 * n_2$ using the grade school multiplication algorithm has time complexity $O(N_1 N_2)$.

Question: For a given positive integer n, what is the $O(\ )$ time complexity of computing n^n, that is, n to the power n ?

Hints:

- The answer is not $O(nN^2)$, i.e. $O(n\,(log_{10} n)^2)$. Such an answer would ignore the fact(s) that when you multiply n repeatedly by itself, the result has an increasing number of digits.
- Answer this question using arguments and concepts that were covered prior to lecture 15. i.e. Do not use the lecture 15 formal definition of big O here.
- In answering the question, do *not* attempt to use any tricks such as computing n^8 by first computing n^4 and then squaring the result. In fact, such tricks do not speed up the computation of n^n in a big O sense. I will discuss this later in the course, time permitting.
- The number of digits N to represent a positive integer n in base 10 is floor($log_{10} n$) + 1, but to answer the question, you should approximate this formula as $log_{10} n$. See lecture notes 2 pages 5,6 where a similar formula is derived for base 2.

d) (10 points)

Let $t(n) = \sqrt{n^2 + 100\,n} - n.$

Use the formal definition of $O(\,)$ from lecture 15 to show that $t(n)\ is\ O(1)$.

What to submit ?

For Question 2(a-d), submit one PDF file **FirstnameLastname.pdf** to the A2Q2 mycourses assignment folder. It will be graded separately from Question 1.

We do *not* require that you typeset your solution. If you do typeset it for fun, that would be fine, and we do encourage you to learn to use typesetting software such as latex. However, learning how to use this software may not the best way for you to spend your time these days.

We do insist that you submit a PDF, though. For example you could do the following:

- write out the solutions neatly by hand and scan them into a single PDF using McGill uPrint
- take a cell phone photo of each page, and insert the images into an MS Word or OpenOffice doc, and save the doc as a PDF.

All symbols and numbers in your document must be clearly legible. Moreover, the logical reasoning of your proofs must be correct and clearly stated. The point of these questions for you to demonstrate that you understand and can express the steps involved. The level of explanations in your solutions should be comparable to the examples in the lecture notes and the exercises.