# Computer Science 331 — Winter 2016
# Assignment #2 Solutions

## Instructions

This assignment concerns lecture material that was introduced in this course on or before Monday, February 8.

Please **read the entire assignment** before you begin work on any part of it.

This assignment is due by 11:59 pm on Friday, February 29.

There are a total of 100 marks available, plus 10 bonus marks.

## Questions

In the language Lisp, each of the four basic arithmetic operators appears before an arbitrary number of operands, which are separated by spaces. The resulting expression is enclosed in parentheses. The operators behave as follows:

- `(+ a b c ...)` returns the sum of all the operands, and `(+)` returns 0.
- `(- a b c ...)` returns `a - b - c - ...` and `(- a)` returns `-a`. The minus operator must have at least one operand.
- `(* a b c ...)` returns the product of all the operands, and `(*)` returns 1.
- `(/ a b c ...)` returns `a / b / c / ...` and `(/ a)` returns `1/a`. The divide operator must have at least one operand.

You can form larger arithmetic expressions by combining these basic expressions using a fully parenthesized prefix notation. For example, the following is a valid Lisp expression:

```
(+ (- 6) (* 2 3 4) (/ (+ 3) (*) (- 2 3 1 )))
```

This expression is successively evaluated as follows:

```
(+ (- 6) (* 2 3 4) (/ 3 1 -2))
(+ -6 24 -1.5)
16.5
```

1. **(15 marks)** Describe an algorithm (pseudocode is *not* required) that uses a stack to evaluate a Lisp expression composed of the four basic operators and integer values. Your algorithm must scan the expression exactly once, and each character may be pushed and popped from the stack only once. When describing how the various operators are processed, you need only include "+" and "-," as the other two operators are identical to these.

   **Solution:** The algorithm initializes an empty stack $S$ and then scans each character in the expression from right to left. For each character $c$, do the following:

   - If $c \in \{), +, -, *, /, 0, 1, 2, \ldots, 9\}$ push $c$ onto the stack.
   - If $c =$ (, we proceed as follows:
     - If the stack is empty or $S.top() \notin \{+, -, *, /\}$, the expression is invalid (last character before a left bracket must be an operator in a valid expression).
     - Let $op = S.pop()$.
       - If $op = +$, set $v = 0$ and pop values off the stack until ) is obtained. If the stack is empty before attempting pop, or a non-numeric value is obtained, report that the expression is invalid. Otherwise, add the new value to $v$. After popping ), push $v$ onto the stack.
       - If $op = -$, then report an invalid expression if the stack is empty or if $S.top()$ is not a number (the $-$ operator requires at least one numeric operand). Then, set $v = S.pop()$. If $S.top() =$), push $-v$ on the stack. Otherwise, proceed as with $+$, subtracting subsequent numerical values from $v$.
       - If $op = *$, proceed analogously to $+$.
       - If $op = /$, proceed analogously to $-$.

       Note: the effect of this step is that the expression $(op\ a_1\ a_2 \ldots a_k)$ is replaced by its value on the stack.
   - Otherwise, the expression contains an invalid character, and we report that the expression is invalid.

   After all characters in the expression have been processed, we pop the stack. If the result is not a numeric value or the resulting stack is not empty, then we report that the expression is invalid. Otherwise, the value obtained is the result of evaluating the expression.

2. **(15 marks)** Write an interface for a `BoundedStack` ADT that extends the interface `cpsc331Stack` provided on the course web page. Your interface should include the following additional operations:

   - `size` — returns the number of elements currently on the stack
   - `capacity` — returns the maximum number of elements the stack can store
   - `isFull` — returns true if the number of elements in the stack is equal to the stack's capacity, false otherwise

   In addition, your interface must declare a revised version of `push` that has the precondition of throwing a `FullStackException` if it is called when the stack is full.

   Notice that the `cpsc331Stack` interface is generic, so your interface must be as well.

Be sure to thoroughly document your interface using the `javadoc` style. You can use the documentation in `cpsc331Stack.java` interface provided as a guide. The quality of your documentation will be taken into account in your grade for this question.

**Solution:** see `BoundedStack.java` on the course web page.

3. **(15 marks)** Write a class that implements your `BoundedStack` interface using an array-based implementation. The constructor to your class should accept one integer parameter to be used as the capacity of the stack. Note that you will need to implement a public class called `FullStackException` that extends `RunTimeException` to be used with this class. The class `EmptyStackException` is part of the `java.util` package.

   Be sure to document your class thoroughly using `javadoc` tags and assertions as appropriate. Both the quality of your implementation and documentation will be taken into account when grading this question.

   **Solution:** see `BoundedArrayStack.java` and `FullStackException.java` on the course web page.

4. **(15 marks)** Write a class that implements your `BoundedStack` interface using a linked list based implementation. The constructor to your class should accept one integer parameter to be used as the capacity of the stack. Be sure to document your class thoroughly using `javadoc` tags and assertions as appropriate. Both the quality of your implementation and documentation will be taken into account when grading this question.

   **Solution:** see `BoundedLinkedStack.java` on the course web page.

5. **(20 marks)** Write a Java program that evaluates Lisp expressions using the algorithm you devised in Question 1. Use your `BoundedStack` interface as part of the implementation of your expression evaluating algorithm. Your program should be called `A2Q5.java`, and the syntax to run it must be

   ```
   java A2Q5 type infile
   ```

   If `type = 0`, your program should use the array implementation of stack, and use the linked list implementation otherwise. The second command-line parameter, `infile`, is the name of an input file containing one Lisp expression (as described above) per line (see, for example, the sample input file `A2Q5.input` on the course web page). For simplicity, you may assume that the integer values in the input expression are single digits in the set $\{0, 1, \ldots, 9\}$, i.e., single non-negative decimal digits; **programs that handle arbitrary integer values will receive 10 bonus marks**.

   The program should print the value obtained by evaluating each expression to standard output, again one per line. Note that, although the operands in the input expression are integers, the output may not be — use `double` for the output values. If the expression is invalid, your program should output `Invalid Expression` and continue to evaluate the next expression in the input file (i.e., it should not terminate in this case). For example, see the sample output file `A2Q4.output`.

**Note**: as part of grading, the Unix `diff` command will be used to compare your program's output to that of a correct program, so be sure to follow these instructions precisely. In particular, send all output to `System.out` and do not output anything except the expression evaluations or the error message `Invalid Expression` (eg. no output headers, etc...). As an example, the output of your program when using the input file `A2Q5.input` should be *exactly* the same as the contents of the file `A2Q5.output`.

**Solution:** see `A2Q5.java` and `InvalidExpressionException.java` on the course web page.

6. **(20 marks)** Implement a test suite (using JUnit) in a file `A2Q6.java` that tests your expression evaluation algorithm as thoroughly as possible. Be sure to include both black box and white box tests. Each test contained in your test suite should contain (as documentation):

   - test input,
   - expected output,
   - purpose of the test.

   Note that the quality of your test suite will be assessed for this question. Provide a written justification that your test suite provides thorough testing of the algorithm.

   **Solution:** The test cases listed here are included in the JUnit program `A2Q6.java`. There are also included in the input file `A2Q6.java`, along with the expected output in `A2Q6.output`. These files were used to test your solution to the previous question.

   Sample black box tests:

| Input | Output | Purpose |
|---|---|---|
| `a` | exception | boundary (invalid char) |
| `+` | exception | boundary (invalid expression) |
| `(` | exception | boundary (invalid expression) |
| `)` | exception | boundary (invalid expression) |
| `()` | exception | boundary (invalid expression) |
| `(+)` | 0 | boundary (min case for $+$) |
| `(*)` | 1 | boundary (min case for $*$) |
| `(-)` | exception | boundary (min case for $-$) |
| `(/)` | exception | boundary (min case for /) |
| `(+ 2)` | 2 | boundary (one operand for $+$) |
| `(* 3)` | 3 | boundary (one operand for $+$) |
| `(- 5)` | $-5$ | boundary (one operand for $+$) |
| `(/ 7)` | 0.1142857... | boundary (one operand for /) |
| `(+ 1 2)` | 3 | typical (two operands for $+$) |
| `(- 7 9)` | $-2$ | typical (two operands for $-$) |
| `(* 3 5)` | 15 | typical (two operands for $*$) |
| `(/ 8 4)` | 2 | typical (two operands for /) |
| `(+ 1 2 3)` | 6 | typical (multiple operands for $+$) |
| `(- 2 3 1)` | $-2$ | typical (multiple operands for $-$) |
| `(* 2 3 4 4)` | 96 | typical (multiple operands for $*$) |
| `(/ 8 2 2 3 5)` | 0.133333... | typical (multiple operands for /) |

Similar cases for the other operations can be devised. Some others:

- `(+ (- 6) (* 2 3 4) (/ (+ 3) (*) (- 2 3 1 )))` — typical example, output 16.5
- `(+ 3 (/ (+ 7 (/ (+ 9 6 (/ (+ 1 (/ (+ (* 4 (+ 1 (* 8 9))) (/ (+ 1 1)))))))))))` — typical example, output approximation of $\pi$
- `(+ 1 (/ 1 (+ 2 (/ 1 (+ 2 (/ 1 (+ 2 (/ 1 (+ 2 (/ 1))))))))))` — typical example, output approximation of $\sqrt(2)$
- `(+ (- 6) (* 2 3 4) (/ (+ 3) (*) (- 2 3 1 ))` — typical example (parentheses unbalanced), output exception
- `(+ - 6) (* 2 3 4) (/ (+ 3) (*) (- 2 3 1 ))` — another typical example (parentheses unbalanced), output exception

The following white-box test cases make sure that every case of the if-statements not covered by the black-box tests is taken at least once, and that every try-catch block is triggered in loops initially and during their execution. Note that these are defined relative to the model solution. Also, note that the `FullStackException` and the `NumberFormatExcepetion` at the end cannot occur, due to the way the algorithm is implemented (stack size is equal to number of characters in the expression, and testing for an initial "(" ensures that at the end the value on the top of the stack will be numeric). The `ArithmeticException` also cannot occur because the operands work on `Doubles`, so division by zero yields `Infinity` instead of an error.

| Input | Output | Purpose |
|---|---|---|
| `(+ 3 a)` | exception | else branch of main if-elseif-else in try block (invalid char) |
| `(5)` | exception | else branch of ( case (operand expected) |
| `(+` | exception | $S$ initially empty for + |
| `(+ - 3 2)` | exception | invalid char on stack for + |
| `(+ 3 4` | exception | $S$ empty in while loop for + |
| `(+ 3 4 - 2 3)` | exception | $S.top$ invalid in while loop for + |
| `(5 (+ 1 2))` | excpetion | $S$ not empty at the end |

Cases for the other operations are analogous (and not included).

The various loops can be made to iterate any number of times by including an expression with the appropriate number of operands, as with some of the black-box test cases.