MOD002702 SOFTWARE IMPLEMENTATION 2017-18
AUTUMN TRIMESTER
MULTIPHASED COURSEWORK

IN-CLASS EXERCISES (10%)

During the semester, weekly practical classes will include a number of short programming exercises (typically 8) that should be demo'd to the tutor in-class (the choice and number of exercises to be assessed is at the discretion of the tutor at the local point of delivery). Each successful demo is worth one mark. For example if 8 exercises are specified for assessment but only 6 are successfully demo'd, you will be awarded 6/8 th's of 10% towards the final mark. NB no work should be submitted for this component, only demo'd to the tutor during scheduled practical class time.

MAIN COURSEWORK ASSIGNMENT (90%)

INFIX TO POSTFIX EXPRESSION TRANSLATOR

In arithmetic the symbols for addition, subtraction, multiplication and division are called *operators*, and the numbers they process are called *operands*. In standard algebraic notation (also known as *infix* notation) the following rules of associativity are used;

1. Operations are performed left to right
2. Multiplication and division are performed before addition and subtraction
3. Expressions inside brackets are evaluated first.

For example, consider the following expression;

A * (B + C) / D

The '*', '+' and '/' are the operators and the 'A', 'B', 'C' and 'D' are the operands. For example if A=3, B=7, C=5 and D=4 then the expression would evaluate out as 9. Clearly the brackets are important as one has to work out the order of precedence of the operations in order to correctly calculate the answer. This infix notation is easy to read and so algebraic computer code is written this way in most programming languages, and input key presses to modern hand-held calculators also support this notation. However the *evaluation* of such an expression usually involves changing it into a form known as *postfix* notation (also known as *reverse polish*). In this notation, operators are written *after* their operands. The advantage of this is that the order of evaluation of operators is always left to right *without* the need for brackets. For example the equivalent postfix expression to the above is;

A B C + * D /

The translation from an infix expression to a postfix expression normally makes use of a dynamic data structure called a stack, constructed to store and retrieve some of the symbols as necessary. The postfix expression can then be evaluated from left to right with each operator acting on values immediately to the left of them, so in this case '+' acts on B and C, '*' acts

on that sum and A, and '/' acts on that product and D. NB Evaluation of a postfix expression is NOT a requirement of this assignment, only it's construction.

The pseudocode algorithm to convert an infix expression into an equivalent postfix expression is as follows;

```
Scan the expression once from left to right.
While (last symbol not processed)
{
    Scan the next symbol, S
    If S is an operand
        Append it to the postfix expression
    Endif
    If S is a left bracket
        Push it onto the stack
    Endif
    If S is a right bracket
        Pop and append all symbols from the stack until the most recent left bracket.
       Pop and discard the left bracket
    Endif
    If S is an operator
        Pop and append to the postfix expression every operator from the stack that is
        above the most recently scanned left bracket and that has precedence greater
        than or equal to the new operator.
        Push the new operator on the stack
    Endif
}
Pop and append to the postfix string everything from the stack
```

Write a program that converts a standard algebraic (infix) expression into an equivalent reverse polish (postfix) expression. Your program should be able to deal with the '+', '-', '*' and '/' arithmetical operators, and left '(' and right ')' brackets.

It is recommended you design an 'Expression' class that stores the infix and postfix strings. That class should include the following methods:

getInfix: Stores the infix expression.
showInfix: Outputs the infix expression.
showPostfix: Outputs the postfix expression

Some other methods that you might need are the following:

convertToPostfix: Converts the infix expression into a postfix expression. The resulting postfix expression is stored in postfixString.
precedence: Determines the precedence between two operators. If the first operator is of higher or equal precedence than the second operator, it returns the value true, otherwise it returns the value false.

You may also need a 'Symbol' class that represents the node of a stack dynamic data structure that your program should build, and optionally a 'Stack' class (depending on how you design/construct the relationship between the Expression class and the stack data structure).

On running the program, a console message should request the typing in of an infix expression, after which the answer should then be displayed to screen in the following form (emboldened text represents an example expression input by the user, followed by pressing the return key);

Enter infix expression: **A+B–C**
Infix expression: A + B – C;
Postfix expression: A B + C –

Test your program on 5 expressions that should range in difficulty of processing. A simple infix expression would be:

A – B

A complicated infix expression would be:

A + ((B +C) * (E – F) – G) / (H – I)

Assume that the expressions you process are error free (ie you need not test for illegal/erronous input expressions).

You may find web-based solutions to this problem, but note they are usually written in a non-object oriented way and adopting such a procedural approach would lose you marks; your program needs to be object-oriented (uses appropriate classes and methods that minimizes the dependence on a main method). Any code which you do use from the public domain must also be properly attributed (referenced in the code itself by suitable comments and relevant reference sources such as URL's or other citations included in the report).

SUBMISSION AND ASSESSMENT

This work should be submitted by the *end* of teaching Week12; for Cambridge this will be to the iCenter by 2pm on **Friday 15th December, 2017** (or by local arrangement for associate/partner colleges). The report should include the following>

i) An 300 word evaluation of your approach to the problem (ie how you designed your program and what functionality you have been able to implement and what could be done better).
ii) A class diagram of your program.
iii) A Test Plan and Results table listing the results of testing with 5 various infix expressions to illustrate functionality and any limitations.
iv) Instructions on how to run your program.
v) A listing of the C# source code.
vi) A hard copy of the report including all documentation as listed above.
vii) An electronic copy of the report including the C# source code and also a compiled executable copy of the program on disk or USB stick.

For details of marking see the feedback sheet.

REASSESSMENT DETAILS

If you are unfortunate enough to require a re-assessment for this module coursework (normally as second attempt but possibly as a first attempt), you should re-submit the main coursework assignment (the expression translator) and your mark out of 100% will be calculated on this new submission only (usually the system will cap this at 40%); note there will be NO in-class exercise component taken into account irrespective of how well that component was undertaken at the first attempt. You are encouraged to explore a different approach to achieving a solution to the main assignment compared to the first attempt.

MOD002702 feedback mark sheet 2017-18 (Autumn Tri 2017)										
	In-class exercises (not used in resubmission)	**Main Assignment**							**Overall 010 mark**	**Overall module result**
Student	**/10**	**Functionality**	**OO design and code development**	**Documentation**	**Comment**	**%**	**Grade**	**/90**	**%**	**Grade**
	Weighted 0.1 Eg if 5 exercises are successfully demo'd out of 9 then the mark out of 10 is> 5/9 of 10 = 5.6	Reflects how well the program works, especially infix translation success. Other enhancements might be clarity of the interface, option to re-run, display of postfix/stack construction during infix parsing).	Choice of user-written classes and their methods. Implementation of infix to postfix algorithm through use of appropriate methods.	Presentation. Code clarity and layout. Correctness of class diagram. Test Plan and Results. Evaluation of approach including success and limitations.	Personalized comments specific to student submission.	Each component is assigned a grade (F, D, C, B, A) and each maps to a mark (0-100) from which the average is taken.	Main assignment grade	Weighted 0.9	Overall weighted mark	F (<40); D (40-49); C (50-59); B (60 - 69); A (70+)
Sid Number	**Mark**	**Grade**	**Grade**	**Grade**	**Comment**	**Mark**	**Grade**	**Mark**	**Final Mark**	**Final Grade**