# MAU11602 Assignment 0
Due 2026-01-29
Solutions

1. (a) Evaluate the postfix expression `1 2 3 4 * + 5 - 6 * +` using the stack based evaluation method described in lecture, showing the stack contents at each step.

   *Solution:* The evolution of the stack is as follows:

```
1    2    3    4    *    +    5    -    6    *    +

 1    2    3    4   12   14    5    9    6   54   55
      1    2    3    2    1   14    1    9    1
           1    2    1         1         1
```

   The top of the stack at the end is 55, so that's the value of the expression.

   (b) The method given in lecture for parsing postfix expressions involved keeping a running count of the number of integers seen minus the number of operators seen. You don't have to parse the expression above but just list that count.

   *Solution:*

```
     1    2    3    4    *    +    5    -    6    *    +

0    1    2    3    4    3    2    3    2    3    2    1
```

   (c) Do you see a relation between this running count and the stack? Can you prove that this relation holds in general, not just in the example?

   *Solution:* The count is always equal to the size of the stack. Initially, i.e. before reading any input, the count is zero and the stack is empty. Whenever we read an integer we increment the count and also push that integer onto the stack, increasing the size of the stack by one. Whenever we read an operator we decrement the count and pop two numbers from the stack and push one onto it, the net effect of which is to decrease the stack size by one. So the change in the count is always the same as the change in the stack size.

   (d) For simplicity I left out one commonly used arithmetic operation, the unary minus, i.e the operator which takes a single integer as argument and changes its sign. Using a `-` to denote this, as is usual, causes problems due to confusion with the `-` for subtraction. One solution to this problem, adopted in SML, is to use `~` for unary minus. We then need to add a rule to our stack based evaluator to say that when we read a `~` then we pop the top element off the stack and push its additive inverse. We also need to change the parsing algorithm as well though. How should we change it?

*Solution:* A `~` only has one argument, so we don't need a rule for separating its arguments but we need to change the rule for separating the arguments for the operators we already had. The running count should stay equal to the stack size, which means we increment the count as before when we read a `+`, `-`, or `*`, but we leave it unchanged when we read a `~`.

2. Consider the postfix grammar for a language with arithmetic operators and relations, as discussed at the end of Lecture 2, i.e.

```
expr ::= bexp | iexp
iexp ::= iexp ws iexp ws oper | int
bexp ::= iexp ws iexp ws rel
oper ::= "*" | "+" | "-"
 rel ::= "<" | "<=" | "=" | ">" | ">="
```

(a) Prove that for every expression consistent with this grammar the number of integers is the number of operators plus the number of relations plus one.

*Solution:* It suffices to prove that for all $n$ any list of tokens of length at most $n$ which can be parsed as an expression has this property. This we can prove by induction. It's vacuously true for $n = 0$. Assuming it's true for a particular value of $n$ we consider a list of length at most $n+1$. If it's of length less than $n+1$ then it's of length at most $n$ and we can use the induction hypothesis directly. If its of length $n+1$ then we proceed as follows. By the first rule it must be an `iexp` or a `bexp`. If its a `bexp` then, by the third rule, it's composed of two `iexp`'s and a `rel`, plus some whitespace which we can ignore. The two `iexp`'s are of composed of lists of tokens of length less than or equal to $n$ so the inductive hypothesis applies to them and each has a number of integers equal to the number of operators plus the number of relations plus one, so together they have a number of integers equal to the number of operators plus the number of relations plus two, but the whole expression has an extra `rel`, and so the number is one. The argument for an `iexp` is similar, but now we have an extra possibility, that the whole expression is just a single `int`, but that's okay since this also satisfies the condition.

(b) Show that the condition which was shown to be necessary in the previous part is not sufficient by giving a list of tokens which can't be parsed even though the number of integers is the number of operators plus the number of relations plus one.

*Solution:* There are lots of examples. The easiest ways to construct an example involve observing either that any valid expression of length greater than one must end in an operator or relation, or that no valid expression can contain more than one relation. Both of these statements can be proved by the same method as in the previous part. `2 + 2` is an example of an input which violates the first condition and `0 0 0 = =` is an example which violates the second. Another approach is to prove that the running total of integers minus operators minus relations is equal to the stack size, as in the previous problem, and so is non-negative. `+ 1 2 3 +` is therefore an invalid input even though it satisfies all the other conditions we've considered.