# COMS10014 Worksheet 1: Terms

**1. Syntax Rules**

(★) Express the following rules of arithmetic as one or more syntax rules in the form $A \vdash B$, then draw the terms in the rule out as trees:

a. The commutative law for multiplication.
b. The associative law for multiplication.
c. Multiplying with 1 leaves a value unchanged; multiplying with 0 always gives 0.
d. The binomial formula for $(a + b)(a + b)$, using only addition, multiplication and brackets.

(★★) A syntax rule $A \vdash B$ can be used as follows:

- Suppose you have any term $T$ (visualised as a tree), and a subtree in $T$ that matches $A$, with variables matching as follows:
  - Any variable nodes in $A$ can stand for arbitrary subtrees.
  - If the same variable appears twice in $A$, it must stand for the same subtree both times.
- Then you can replace $A$ with $B$ in $T$ to get a new term $U$ as follows:
  - Any variable node that matched a subtree in $A$ must become the same subtree in $B$.
  - Any variable node in $B$ that did not appear in $A$ can become any term you want.

Draw the term $T = 1 + (x + 1)(3 \times y + 2)$ as a tree and then apply the rule $a(b + c) \vdash ab + ac$ 'graphically' to the one node in $T$ which it matches. What terms do $a, b, c$ stand for? What new term $U$ do you get? *Note: the point of this question is to visualise what 'applying a syntax rule' means on a term represented as a tree.*

**2. RPN**

(★) Give four rules for printing terms in RPN, following the schema of rules I1–I4 in the lecture notes. You do not need brackets anywhere. To make things simpler, assume that after every token (value, variable name, or operation symbol) you print a space, and that the symbol for unary minus is ~ to distinguish it from the symbol −.

(★) In a term string in RPN, how do you find the top-level operation (if you draw the term as a tree, this means the root node)?

(★★) Convert the terms on the left to RPN, and the terms on the right to infix:

$3 + 2 - 5 - 1$

$(a + b) \times (a - b)$

$\dfrac{a/2 + 1}{b/2 - 1}$

$b/2 - \sqrt{b^2 - 4ac}$
where squaring and taking the square root are unary operations

$2\ 1 + 3\ 6 \times +$

$3\ a\ a \times \times 2\ a \times + 5 +$

$a \sin^2 b \cos^2 +$
where squaring and taking the sine and cosine are unary operations

$a\ k\ \hat{}\ 1 - a\ 1 - /$
where ^ (exponent) is a binary operation

**3. Brackets**

(★★★) When you are printing an arithmetic term in infix notation, when are brackets necessary? There are actually two kinds of 'necessary' – can you give an example of each?

1. Cases where, if you forget the brackets, you get a term string that evaluates to a different value than the original term.
2. Cases where, if you forget the brackets, you get a term string that parses to a different term than the original term, even though it evaluates to the same value.

Give a version of rule I4 (from the lecture notes) that prints only the necessary brackets. You can ignore unary operations in rule I3 (assume that they always need brackets).

You might want to consider terms with just + and × operations to start with; your rule I4 can then treat the two cases differently. Afterwards, you can think about – and ÷ operations too.

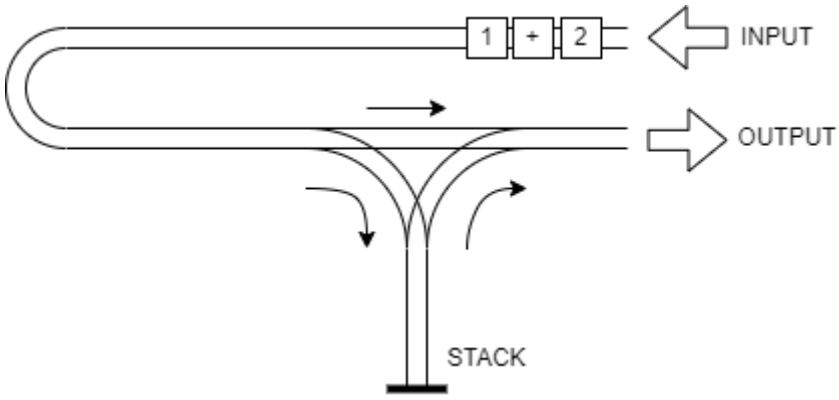
**4. Infix to RPN**

Here is a procedure for turning strings in infix notation into reverse Polish notation. This immediately gives a way to parse term strings in infix notation into terms:

1. Convert the term string to a list of tokens: numbers, variable names, operators, or brackets (this is easy, basically if you see a digit[1] then it and all following digits become one token, skip spaces, anything else is a token of its own).
2. Run this list through the procedure to produce a list in RPN.
3. Run the RPN list through a RPN parser to produce a term-as-a-mathematical-object (using whatever data types your programming language supports).

We ignore unary operations for now to make the exercise a bit easier.

It is helpful to imagine a railway junction like this, with each token on its own wagon:




Tokens enter the junction at the 'input' track. The bend before the junction is there so we can write the tokens left to right, as we are used to, but make sure the leftmost (not the rightmost) token enters the junction first (the string to parse here is $1 + 2$ not the reverse $2 + 1$).

We can assume the stack is large enough to hold as many tokens as we need.

[1] For negative numbers, then a minus sign immediately in front of a digit becomes part of the same token as the digit and all following digits.

For each token, the procedure can do one of three things:

1. Send a token from input to output.
2. Send a token from input to the stack.
3. Send one or more tokens from the stack to the output, then do one of the above.

The point of the stack is that it operates a 'last in, first out' policy: if you send first a token '1' then a token '2' to the stack, then (in our drawing with the stack facing downwards) the '2' token ends up on top of the '1' token, so if you later ask to send a token from the stack to the output, this always refers to the top token on the stack.

The procedure, ignoring brackets for now, reads tokens one by one from the input.

a) Value and variable tokens from the input go directly to the output.
b) Operator and bracket tokens are handled using the stack.
c) Once all input tokens are read, if there are operator tokens left on the stack, they are sent to the output.

Part a) means that variable and value tokens will always end up in the same order in the output as they appeared in the input, which is the correct behaviour: for example, $1 + 2 \times 3$ becomes $1\ 2\ 3\ \times\ +$ in RPN; the order of operators may have changed but the values are in the same order as they were in the infix formula.

(★★) Q 1: complete the following rule for handling operator tokens, assuming no brackets.

- When you read an operator token from the input:
  a. ???
  b. Send the token from the input to the stack.

Hint: as we are currently parsing term strings without brackets, what two situations (from exercise 3.) cannot happen, because brackets would be required to print such a term?

Execute your procedure (on paper) on the following test cases.

| **Input** | **Output** | **Notes** |
|---|---|---|
| `1 + 3` | `1 3 +` | |
| `6 - 2 - 1` | `6 2 - 1 -` | Not: `6 2 1 - -` as you want the result 3 not 5. |
| `1 × 2 + 3` | `1 2 × 3 +` | |
| `1 + 2 × 3` | `1 2 3 × +` | Note the difference to the example above. |
| `1 + 2 + 3 + 4` | `1 2 + 3 + 4 +` | |
| `1 + 2 × 3 + 4` | `1 2 3 × + 4 +` | |
| `1 + 2 + 3 × 4` | `1 2 + 3 4 × +` | |
| `1 × 2 + 3 × 4` | `1 2 × 3 4 × +` | |

(★★) Q 2: Now, think about how the procedure needs to be extended to allow brackets. The procedure now can do one more thing:

4. Delete a token from the input and/or the stack. Then do one of the rules 1.–3.

Hint: you will want to delete all bracket tokens at some point, as brackets never appear in RPN term strings (or terms themselves, as abstract mathematical objects). But, you need to use the

stack to keep track of *opening* brackets that you have seen so far, so you can react correctly when a matching closing bracket comes along.

- What do you have to do when you see a bracket token (there are two cases, for opening and closing brackets)?
- What change(s) do you have to make to the "send all the stack to the output" part for when the input is finished?
- How can the procedure detect the following – in all these cases, the input was not a valid term string, so you can stop immediately if you detect them:
  a) An opening bracket without a matching closing bracket.
  b) A closing bracket without a matching opening bracket.
  c) Some other bracket mismatch, for example the string `) 1 + 2 (`.

Here are some more test cases for you:

| **Input** | **Output** | **Notes** |
|---|---|---|
| `1 + (2 + 3)` | `1 2 3 + +` | With bracketed input, we can have two + in a row in the output. |
| `(1 + 2) × 3` | `1 2 + 3 ×` | |
| `(1 + 2) + 3` | `1 2 + 3 +` | These brackets are redundant, but allowed. |
| `1 × (2 + 3)` | `1 2 3 + ×` | |
| `1 + 2 × 3` | `1 2 3 × +` | This test case appeared before in the exercise without brackets – make sure it still works. |
| `(1 + 2) × (3 + 4)` | `1 2 + 3 4 + ×` | |