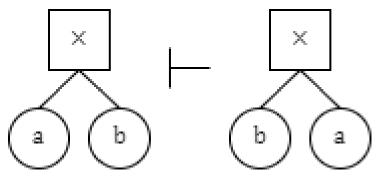
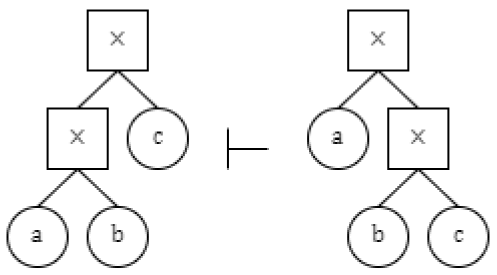
COMS10014 Solutions 1: Terms

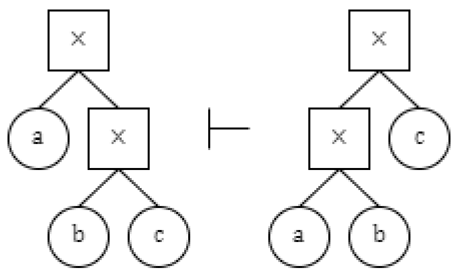
1. Syntax Rules

The commutative law of multiplication is $a \times b \vdash b \times a$ or as trees,



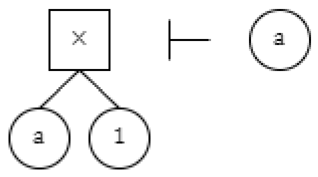
The associative law of multiplication is $(a \times b) \times c \vdash a \times (b \times c)$ and $a \times (b \times c) \vdash (a \times b) \times c$:

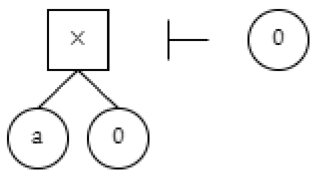




We need two rules here to capture both directions. For the commutative law, we did not need $b \times a \vdash a \times b$ as that is the same rule as $a \times b \vdash b \times a$ with just the variable names reversed, whereas for the associative law, the two sides of the rule have different tree structures.

Multiplying with 1 resp. 0 is $a \times 1 \vdash a$ and $a \times 0 \vdash 0$:



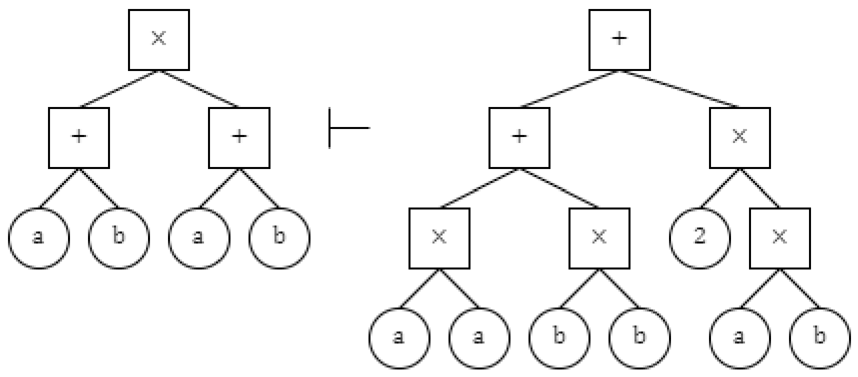


The binomial formula:

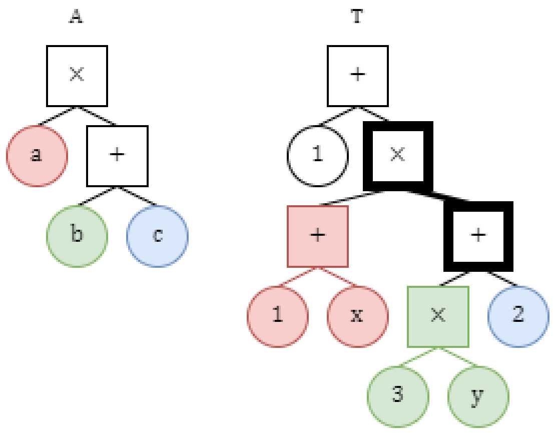
$$(a+b)(a+b) \vdash a \times a + b \times b + 2 \times (a \times b)$$

$$a \times a + b \times b + 2 \times (a \times b) \vdash (a+b)(a+b)$$

As trees:

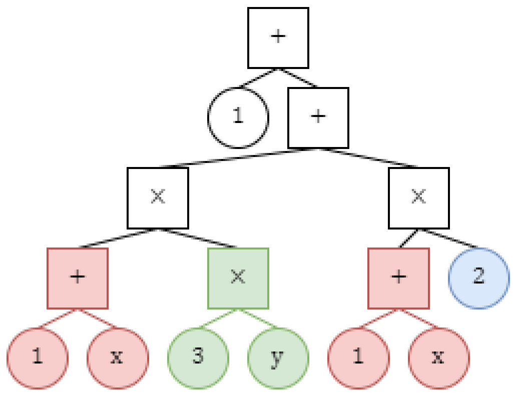


For the terms A and T,



the term A is at its root a multiplication node, whose right child is an addition node. There is exactly one such instance in T (highlighted). So, if we set a, b, c to be the respective subtrees underneath the matching pattern, we get $a \coloneqq 1 + x$, $b \coloneqq 3 \times y$ and $c \coloneqq 2$.

Substituting, we get $U =$



which is

$$1 + \big((1 + x) \times (3 \times y) + (1 + x) \times 2\big)$$

We assume that you are already competent in the mathematics here, as you have done A-level mathematics or an equivalent course already. What is new, and the point of this exercise, is connecting what you already know 'algebraically' to the visual / structural tree representation.

2. RPN

To print a term in RPN:

1. If a term is a value term, print the value followed by a space.
2. If a term is a variable term, print the variable name followed by a space.
3. If a term is a unary operation term with child T, print T, then print the operation symbol followed by a space.
4. If a term is a binary operation term with children T, U, then
   a. Print T.
   b. Print U.
   c. Print the operation symbol, followed by a space.

We do not need to worry about the space between T and U in Rule 4, as the command 'Print T' will already deal with that space already: all four rules, covering all possible terms, all end with 'followed by a space'.

The one thing that we have glossed over is how to represent negative numbers, as you cannot write -2 without confusing the RPN parser as described in the lecture notes. Either you need yet another symbol for this, or you use unary minus to represent this, so -2 would be printed as the string '2 ~ ' (without quotes).

In RPN, the top-level operation is always the last (rightmost) one in the string.

For the conversion question,

Infix	**RPN**	**RPN**	**Infix**
$3+2-5-1$	$3\ 2 + 5 - 1 -$	$2\ 1 + 3\ 6 \times +$	$(2+1)+3\times 6$
$(a+b)\times(a-b)$	$a\ b + a\ b - \times$	$3\ a\ a \times \times\ 2\ a \times + 5 +$	$3\times a\times a+2\times a+5$
$\dfrac{a/2+1}{b/2-1}$	$a\ 2\ /\ 1 + b\ 2\ /\ 1 - /$	$a \sin\ {}^2\ b \cos\ {}^2\ +$	$\sin(a)^2+\cos(b)^2$
$b/2-\sqrt{b^2-4ac}$	$b\ 2\ /\ b\ {}^2\ 4\ a\ c \times\times - \sqrt{}\ -$	$a\ k\ \hat{}\ 1 - a\ 1 - /$	$\dfrac{a^k-1}{a-1}$

3. Brackets

We start with terms containing only $+$ and $\times$ binary operations. The first insight is that because of 'multiplication before addition', we do not need brackets if a $+$ node has a $\times$ child, as this is parsed the right way round already (in the term $1\times 2+3$, the $+$ is the top-level operation, draw the term as a tree if this is not clear to you). However, we do need brackets if a $\times$ node has a $+$ child, as in $1\times(2+3)$ the $\times$ is the top-level operation.

The next insight is that to handle this, we need to move the responsibility for printing brackets around a term up to its parent term. Thus, a first attempt at a new rule I4 is as follows:

I4a. If the term is a $\times$ operation term with children T, U, then

1. If the first child T is a $+$ term, then print an opening bracket, then print T, then print a closing bracket. If not, just print T.
2. Print the operation symbol $\times$.
3. If the second child U is a $+$ term, then print an opening bracket, then print U, then print a closing bracket. If not, just print U.

Otherwise, just print T, then print the operation symbol $+$, then print U.

This is enough to get all brackets where forgetting them would change the evaluation of the term, in terms with only $+$ and $\times$. However, the terms $(1+2)+3$ and $1+(2+3)$, although they evaluate to the same value, are not the same as terms (which is why we need an associative law if we are working purely with syntax). Draw the trees of both terms if this is not clear to you. The answer now depends on which of the two $1+2+3$ is meant to mean; the programming language C defines it to mean the version with brackets on the left. C does the same with other arithmetic operators, so $2-3-5$ means $(2-3)-5$ which is both consistent with mathematics as you learnt it at school and makes a difference to the evaluation because the – operation, unlike $+$, is not associative.

These ideas are in fact enough to deal with all four basic operations of arithmetic. Thus, brackets around a child term are required if and only if either of the following cases occurs:

1. The child term has lower precedence than the term we are currently printing (where $\times$,$\div$ are both 'high' precedence and $+. -$ are both 'low').
2. The *right* child term of an operation term is another operation term of the same precedence (since such terms without brackets are parsed left-first).
   If the operation is associative then leaving the brackets off does not affect the evaluation, but does give a different term (as a tree); if the operation is note associative then this can affect the result that the term evaluates to as well.

So the full rule is

I4b. To print a binary operation term with children T, U:

1. If the left child T is an operation term with lower precedence (that is, the term we are printing is a $\times$ or $\div$ and the child is a $+$ or $-$), then print an opening bracket, then print T, then print a closing bracket. Otherwise, just print T.
2. Print the current term's operation symbol.
3. If the right child U is a binary operation term of the *same or lower* precedence, then print an opening bracket, then print U, then print a closing bracket. If not, just print U.

(We assume that unary operation terms always print their own brackets, so we do not need to print brackets for them a second time if their parent term is a binary operation.)

This reasoning is actually part of the background to the usual way of writing a parser for infix terms. Reading from left to right, as long as you do not see any brackets, you need to avoid parsing a term in such a way that an operation term ever has a parent of higher precedence than itself; if this would happen, you backtrack and parse the term 'the other way round'. This means that $1 + 2 \times 3$ gets parsed correctly, like you would expect it to from school mathematics.

4. Infix to RPN

This procedure is called the *shunting-yard algorithm* and it was invented by the famous computer scientist Edsger Dijkstra.

Q 1: the forbidden cases are, for infix term strings *without* brackets,

- An operator node in a term has a child of lower priority than itself (for example, a $\times$ node has a $+$ child).
- An operator node in a term has a right child that is another operator of the same priority. For example, in $1 - (2 - 3)$, the brackets are required as $1 - 2 - 3$ means $(1 - 2) - 3$.

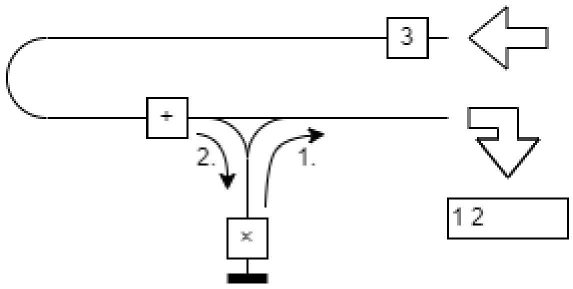
If a RPN string contains the sequence $+\ \times$ of operators then this creates a $+$ node as the right child of the $\times$ node, as in $1\ 2\ 3 + \times$ (draw the tree if necessary to see this). So an infix term without brackets, converted to RPN, can never have a low-priority operator immediately followed by a higher-priority one. Similarly, $1\ 2\ 3 + +$ is a RPN string where the last $+$ has the other one as its left child, unlike $1\ 2 + 3 +$ which is the other way round (if you replace both $+$ with $-$, this matters for evaluation too). So, two operators of the same priority cannot directly follow each other either.

When you read a token from the input, the rule is:

- If the top of the stack contains an operator of the same or higher priority than the current token, send the top token on the stack to the output. Repeat this until either the stack is empty, or the top token is something else (currently the only case possible is a token of lower priority). Then send the current input token to the stack (not the output).

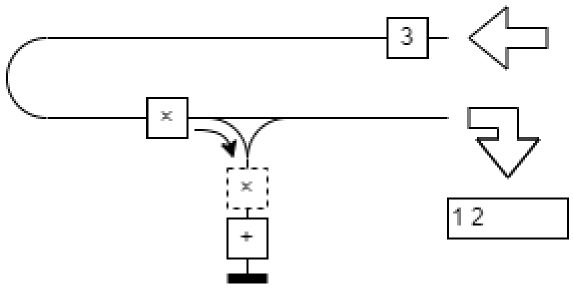
This rule is enough to handle all our test cases correctly.

While parsing $1 \times 2 + 3$, when we reach the $+$ token, the $\times$ is on the stack and the output so far reads $1\ 2$. Because the $+$ is lower priority than the $\times$ on the stack, we first clear the $\times$ off the stack by sending it to the output, then put the $+$ on the stack:



Next, we output the 3, and then because the input is finished, we clear the $+$ off the stack to get $1\ 2 \times 3\ +$.

On the other hand, parsing $1 + 2 \times 3$, when we read the $\times$ then the $+$ on the stack is lower priority, so we do not clear it but put the $\times$ on the stack:



so the stack is now $+ \times$ where the $\times$ was added last. We output the 3, and then clear the stack in last-in-first-out order to get $1\ 2\ 3 \times +$. The stack has had the effect that the two operators appear in the output in the opposite order to the way round they were in the input.

Because you never add an operator to the stack when there is already a higher-priority operator on the stack top, you cannot have a situation where a lower-priority operator comes directly after a higher-priority one in the output (you can produce $\times\ +$ but never $+\ \times$).

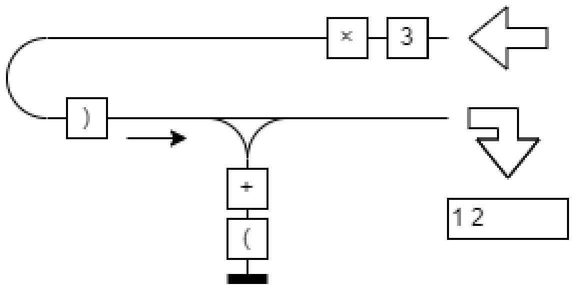
Q 2: We keep track of brackets by putting *opening* bracket tokens on the stack.

- If the input token is an opening bracket, send it to the stack and do not output anything.
- If the input token is an operator token and the top token on the stack is an opening bracket, always send the input token to the stack and do not output anything.
  (Some people write this as ‘brackets have lower priority than operators’.)

- If the input token is a closing bracket, then while the top token on the stack is not a bracket (that is, it is an operator), send the top stack token to the output. Repeat this until the stack does not contain an operator as its topmost token.
    - If the top token on the stack is now an opening bracket, delete both it and the current input token (which is the matching closing bracket).
    - If the stack is empty (you did not find an opening bracket token to delete), then the input string had mismatched brackets, so it is not a valid term string. Specifically, there was a closing bracket without a matching opening bracket.

When the input is empty and you send tokens from the stack to the output, if there are any bracket tokens left, then these are opening brackets without a matching closing bracket, so the input was not a valid term string. Operators work the same as before.

This procedure makes sure that when you open a bracket and close it again, all the operators that lived inside the brackets are processed before you move any further from the closing bracket. For example in $(1 + 2) \times 3$, when you reach the closing bracket you are in this situation:



Before we can delete the closing bracket, we need to output all operations that are on the stack above the last opening bracket, in this case the $+$. This way, the $+$ can arrive at the output before the following $\times$ to get the result $1\ 2 + 3\ \times$.

You detect bracket errors as follows:

a) An opening bracket without a matching closing bracket causes an error when you are clearing the stack at the end of the input: there was nothing to remove the opening bracket, so it will still be around (and you can stop the procedure with an error).
b) A closing bracket without a matching opening bracket will cause the stack to empty while you are removing the stack items ‘until you find an opening bracket’.
c) If you see a closing bracket before its matching opening bracket, the same as in case b) will happen.