

AQA Computer Science A-Level
4.3.3 Reverse Polish
Advanced Notes



Specification:

4.3.3.1 Reverse Polish – infix transformations

Be able to convert simple expressions in infix form to Reverse Polish notation (RPN) form and vice versa. Be aware of why and where it is used. Eliminates need for brackets in sub-expressions. Expressions in a form suitable for evaluation using a stack. Used in interpreters based on a stack for example Postscript and bytecode.





Infix Notation

Humans prefer to use **in-fix** order of notation. This means that the **operand** is **either side** of the **opcode**. However, longer equations can cause confusion over the **order of execution**.

Example 1:

$$3 + 5$$

3 and 5 are the **operand** and + is the **opcode**. The answer is 8.

Example 2:

$$9 + 6 / 3$$

BODMAS

Determines the order of execution of an equation. In order of priority: Brackets, Orders (powers/indices), Division, Multiplication, Addition, Subtraction. According to this rule, the answer to $4 + 8 / 2 - 3$ is 5.

These expressions can either use brackets or **BODMAS** to alleviate the confusion.

According to BODMAS, the following equation is produced.

$$9 + 6 / 3 = 11$$

Synoptic Link

Opcode is the **instruction** (e.g. + / MOV). **Operand** refers to the **data** on which the **operation** is being performed. In $1 + 2$, one and two are the operand, and + is the opcode.

Opcode and Operands are covered in **Structure and role of the Processor and its Components** under **Fundamentals of Computer Organisation and Architecture**.



However, brackets could be added to produce an equation with a different answer.

$$(9 + 6) / 3 = 5$$

Reverse Polish Notation

Reverse Polish Notation (RPN) is a **postfix** way of **writing expressions**. This **eliminates** the need for **brackets** and any confusion over the **order of execution**. Rather than the opcode going in **between** the operand, a postfix expression writes the opcode **after** the operand. When the **opcode** has **both** pieces of **operand immediately preceding** it, the operation proceeds.

Reverse Polish Notation

Also known as postfix notation or RPN. The operators follow the operand.

Example 1:

This is an **infix** equation.

$$3 + 5$$

This is its **postfix** equivalent.

$$3 5 +$$

They both give the answer 8.



Example 2A:

This is an **infix** equation. Its answer is 11.

$$9 + 6 / 3$$

This is its **postfix** equivalent.

$$9 \ 6 \ 3 \ / \ +$$

Proof

The / sign has two pieces of **operand immediately before** it (6 and 3).

$$9 \ 6 \ 3 \ / \ +$$

It performs the operation $6 / 3$, which equals 2.

$$\begin{array}{c} 9 \ 6 \ 3 \ / \ + \\ 6 \ 3 \ / \ = \ 6 \ / \ 3 \ = \ 2 \\ 9 \ 2 \ + \end{array}$$





Now the **postfix** expression reads 9 2 +. The 9 and the 2 are immediately before the plus sign.

$$9\ 2\ +$$

They are added together to make 11, the same as its **infix** equivalent.

$$9\ 2\ +$$

$$9\ 2\ + = 9 + 2 = 11$$

$$11$$

Example 2B:

This is an **infix** equation. Its answer is 5.

$$(9 + 6) / 3$$

This is its **postfix** equivalent.

$$9\ 6\ +\ 3\ /$$



Proof

The + has two pieces of operand preceding it (9 and 6).

$$9 \ 6 \ + \ 3 \ /$$

They add together to make 15.

$$9 \ 6 \ + \ 3 \ /$$

$$9 \ 6 \ + \ = \ 9 \ + \ 6 \ = \ 15$$

$$15 \ 3 \ /$$

The new expression is 15 3 /, which is the same as 15 / 3, hence the answer is 5. This is the same answer as given by the infix equation (9 + 6) / 3.

$$15 \ 3 \ /$$

$$15 \ 3 \ / \ = \ 15 \ / \ 3 \ = \ 5$$

$$5$$



Converting from Infix to Postfix

Infix expressions can be converted into postfix by the **postorder traversal** of an **expression tree**. Simpler ones can be done by observation.

Example 1:

The following expression needs to be converted into its **postfix** equivalent.

$$((y - 6) / 3) * (x + 4)$$

The first operator is selected.

$$((y - 6) / 3) * (x + 4)$$

The minus sign is our first **opcode**. Because of the brackets around the operation, the two pieces of **operand** are 12 and 6. 12 - 6 is the same as 12 6 - in RPN, so this part of the equation can be replaced.

Synoptic Link

Graphs can be used as **visual representations of complex relationships**. A **tree** is an **acyclic connected graph**. An **expression tree** is a **binary tree** with **operand** and **opcode** as **nodes**.

Graphs are covered in **Graphs** under **Fundamentals of Data Structures**. Trees are covered in **Trees** under **Fundamentals of Data Structures**.

Synoptic Link

Traversal is the process of **visiting** each **node** in a **graph**. There are several types, each outputting the nodes in a different order. **Postorder traversal** is exclusively for **trees**.

Traversals are covered in **Graph-traversal** and **Tree-traversal**, both under **Fundamentals of Data Structures**.

Note

This method will work by choosing any operator first, but it is a smart idea to work from left to right as to not forget any part of the expression. Do not remove the brackets until the end.





$$((y - 6) / 3) * (x + 4)$$

$$y - 6 = y \ 6 -$$

$$((y \ 6 -) / 3) * (x + 4)$$

The next **operator** can be looked at.

$$((y \ 6 -) / 3) * (x + 4)$$

It is a divisor. The two pieces of operand surrounding it is 3 and the result of y 6 -.

$$((y \ 6 -) / 3) * (x + 4)$$





This may seem confusing, but remember, $y - 6$ can be evaluated (with a value of y), so it can be treated as a single term.

$$((y - 6) / 3) * (x + 4)$$

$$(y - 6) / 3 = (y - 6) 3 /$$

$$((y - 6) 3 /) * (x + 4)$$

The next operator is observed.

$$((y - 6) 3 /) * (x + 4)$$





The **operand** surrounding the multiplication sign is the result of the **postfix** expression $((y - 6) / 3)$ and the result of the **infix** expression $(x + 4)$. Again, this is less complicated than it looks if each operand is taken as one term.

$$((y - 6) / 3) * (x + 4)$$

$$((y - 6) / 3) * (x + 4) =$$

$$((y - 6) / 3) (x + 4) *$$

$$((y - 6) / 3) (x + 4) *$$

It would be tempting to say that we have found the **postfix** equivalent - there isn't an **operator** to the right of the multiplication symbol. However, if we look back at the original equation, we can see that the $+$ sign needs to be dealt with. Original equation:

$$((y - 6) / 3) * (x + 4)$$

Current equation:

$$((y - 6) / 3) (x + 4) *$$



The **operand** surrounding the plus sign is x and 4.

$$((y \ 6 \ -) \ 3 \ /) \ (x \ + \ 4) \ *$$

$$x \ + \ 4 = x \ 4 \ +$$

$$((y \ 6 \ -) \ 3 \ /) \ (x \ 4 \ +) \ *$$

Now all the opcode has been considered, the **brackets can be removed** as they are superfluous.

$$y \ 6 \ - \ 3 \ / \ x \ 4 \ + \ *$$

Synoptic Link

Stacks are a **data structure** with a **LIFO** (Last In, First Out) **order of execution**. Items **added** onto a stack are said to be "**pushed**". An item is **removed** by "**poping**".

Stacks are covered in **Stacks** under **Fundamentals of Data Structures**.

Stacks

Stacks can be used to **evaluate postfix equations**. The **algorithm** goes along the array - **operand** is **pushed** onto the **stack**, whilst **opcode** causes **two items** to be **popped** off the **stack** with the **result** of the operation **pushed** onto the **stack**.

Algorithm

An **algorithm** is a set of instructions which **completes a task** in a **finite time** and **always terminates**.





Example 1:

The following RPN expression needs to be evaluated:

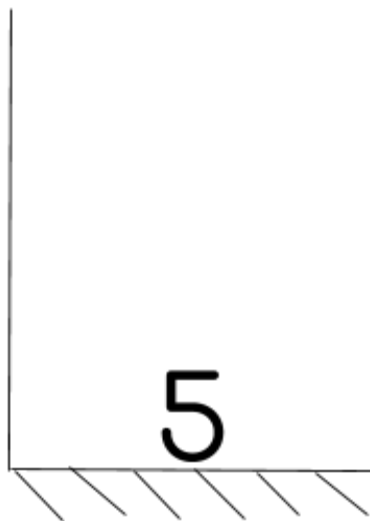
$$5 \ 3 \ - \ 4 \ +$$

The **leftmost** item is selected first.

$$5 \ 3 \ - \ 4 \ +$$

5 is the **operand** so it is **pushed** onto the **stack**.

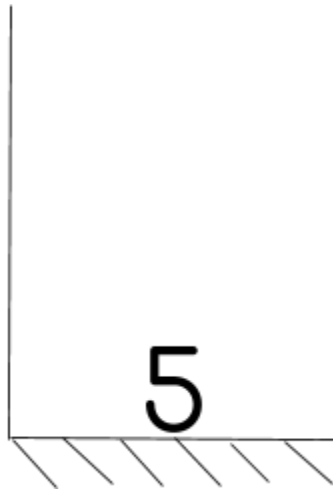
$$5 \ 3 \ - \ 4 \ +$$





The next item is looked at.

$$5 \ 3 \ - \ 4 \ +$$

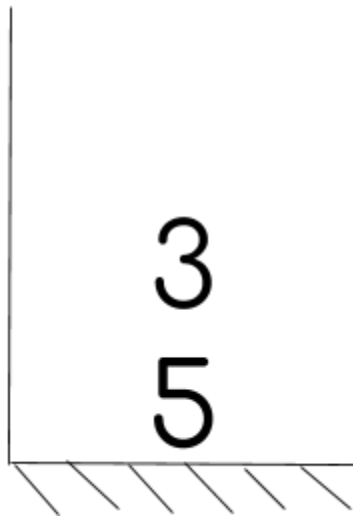


3 is also **operand** so it is **pushed** onto the **stack**.



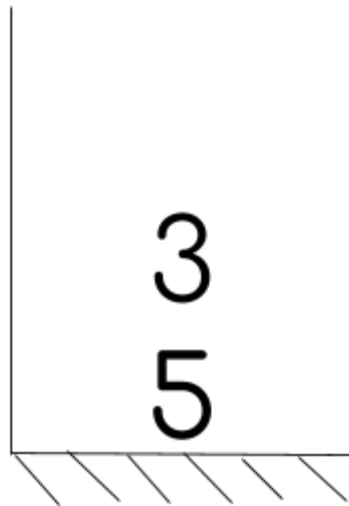


$$5 \quad 3 - 4 +$$



The next item is investigated.

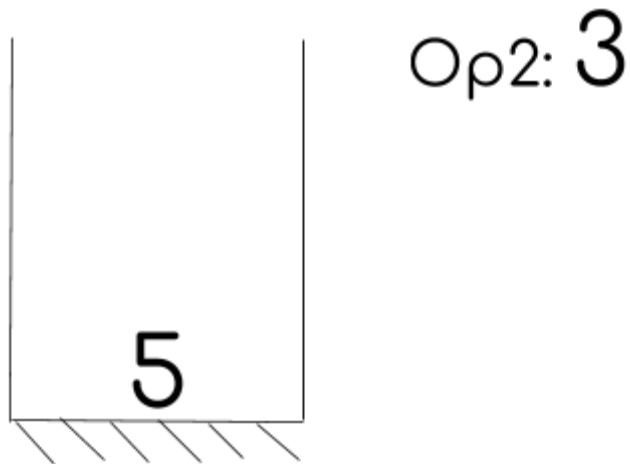
$$5 \quad 3 - 4 +$$





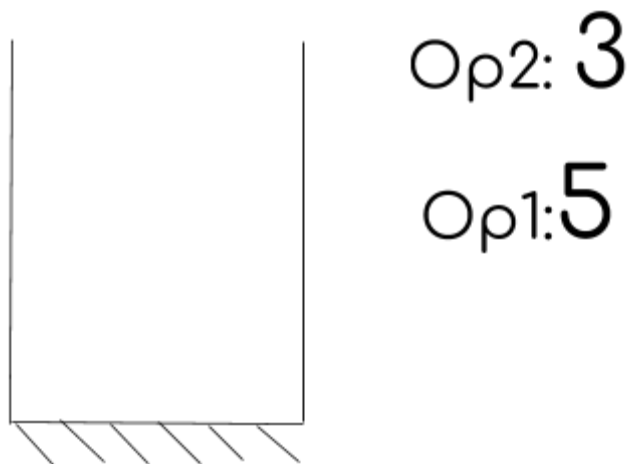
The minus sign is an **operator**. Therefore two items are **popped** off the **stack** - they will be the **operand** for this operation. First pop:

$$5 \ 3 \ - \ 4 \ +$$



The 3 has been labelled as operand 2, this will help show the order of operation. Second pop:

$$5 \ 3 \ - \ 4 \ +$$



Now we have the opcode and the operand, an equation can be evaluated.





Note

Remember the first item to be popped off the stack is the second operand (in infix, it is the operand which goes after the operator).

5 3 - 4 +



Op2: 3

Op1: 5

5 - 3

The result is then pushed onto the stack.

5 3 - 4 +



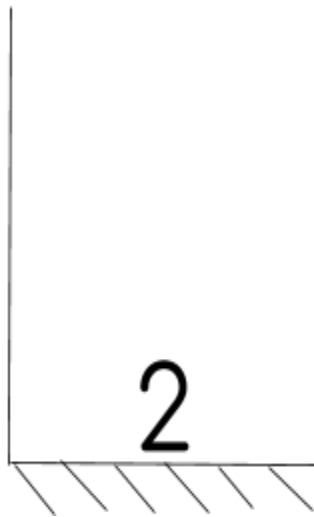
5 - 3 = 2

Now, the next item is looked at.



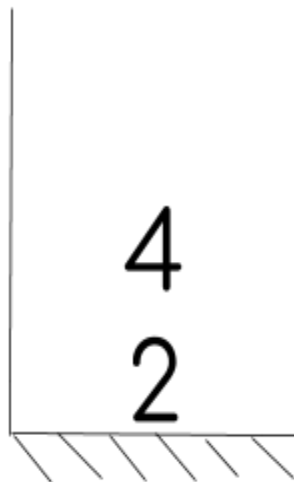


$$5\ 3\ -\ 4\ +$$



4 is the **operand** so it is **pushed** onto the **stack**.

$$5\ 3\ -\ 4\ +$$

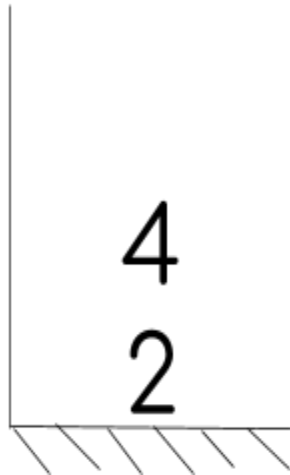


The next item is observed.



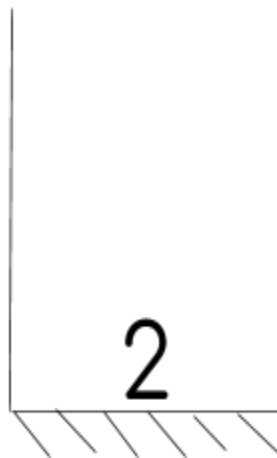


$$5\ 3\ -\ 4\ +$$



The addition sign is an **operator**, so **two** items are **popped** off the **stack**. First pop:

$$5\ 3\ -\ 4\ +$$



Op2: 4

Second pop:





$$53 - 4 +$$



Op2: 4

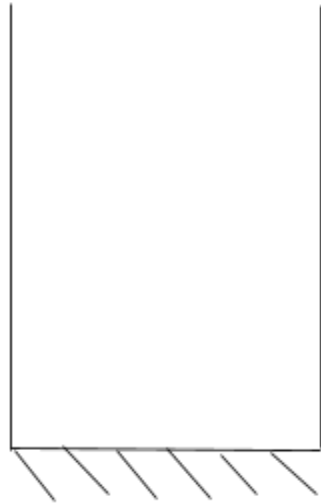
Op1: 2

The operation can now be performed.





$$53 - 4 +$$



$$\text{Op2: } 4$$

$$\text{Op1: } 2$$

$$2 + 4$$

The answer is pushed onto the stack.

$$53 - 4 +$$



$$2 + 4 = 6$$

The next item is considered. There are no more items to consider.



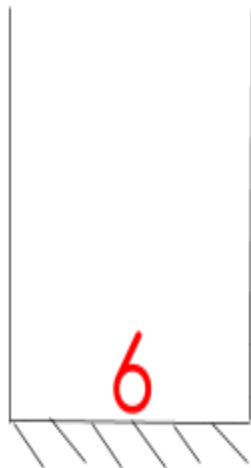


$$53 - 4 +$$



The [top of the stack](#) is returned as the answer. The algorithm [terminates](#).

$$53 - 4 + = 6$$





Example 2:

The following postfix expression needs to be evaluated.

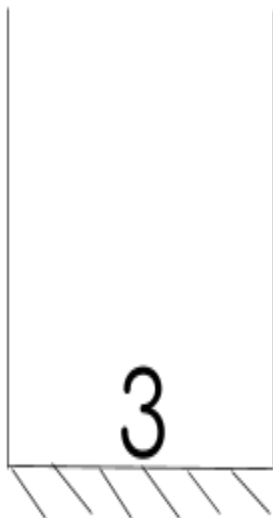
$$3\ 7\ +\ 2\ /\ 5\ *\ 6\ 8\ -\ 2\ * -$$

The leftmost item is selected first.

$$3\ 7\ +\ 2\ /\ 5\ *\ 6\ 8\ -\ 2\ * -$$

3 is the operand so it is pushed onto the stack.

$$3\ 7\ +\ 2\ /\ 5\ *\ 6\ 8\ -\ 2\ * -$$



The next item is considered.





$$3 \color{red}7 + 2 / 5 * 6 8 - 2 * -$$



7 is the **operand** so it is **pushed** onto the **stack**.

$$3 \color{red}7 + 2 / 5 * 6 8 - 2 * -$$



The next item is considered.



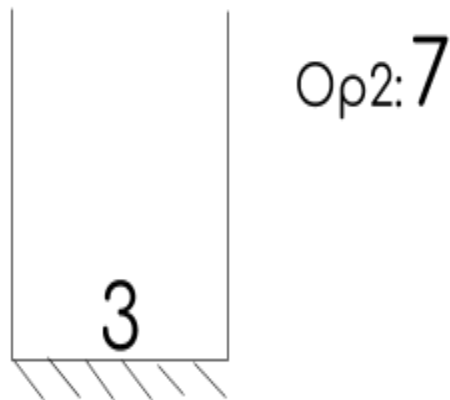


$$37 + 2 / 5 * 68 - 2 * -$$



The plus is **opcode**. Hence, **two** items are **popped** off the **stack**. First pop.

$$37 + 2 / 5 * 68 - 2 * -$$



Second pop.





$$37 + 2 / 5 * 68 - 2 * -$$



Op2: 7

Op1: 3

The operation can now be performed.

$$37 + 2 / 5 * 68 - 2 * -$$



Op2: 7

Op1: 3

3 + 7

The [result](#) of this calculation is then [stored](#) in the [stack](#).





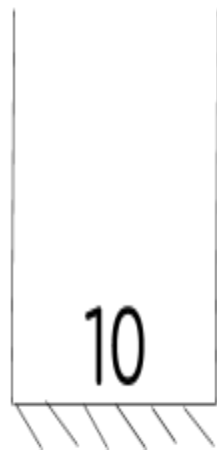
$$37 + 2 / 5 * 68 - 2 * -$$



$$3 + 7 = 10$$

The next item is considered.

$$37 + 2 / 5 * 68 - 2 * -$$



The 2 is the **operand** so it is **pushed** onto the **stack**.





$$37 + 2 / 5 * 68 - 2 * -$$



The next item is looked at.

$$37 + 2 / 5 * 68 - 2 * -$$



The division sign is an **operator** so **two** items on the **stack** are **popped** off. First pop.





$$37 + 2 / 5 * 68 - 2 * -$$



Op2: 2

Second pop.

$$37 + 2 / 5 * 68 - 2 * -$$



Op2: 2

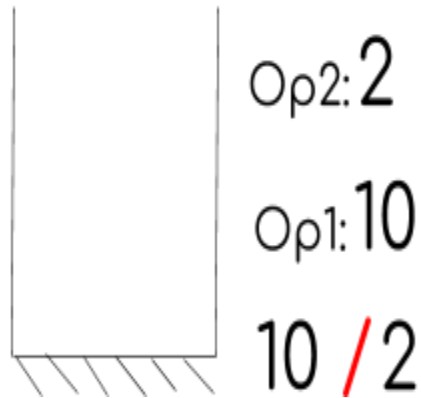
Op1: 10

The equation can be evaluated.



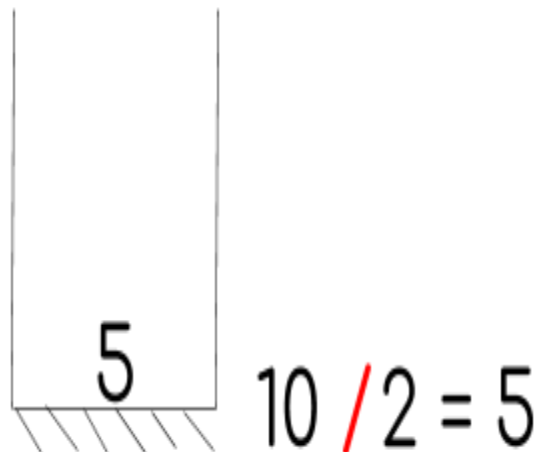


$$37 + 2 / 5 * 68 - 2 * -$$



The result is pushed onto the stack.

$$37 + 2 / 5 * 68 - 2 * -$$



The next item is observed.





$$37 + 2 / 5 * 68 - 2 * -$$



5 is the operand so it is pushed onto the stack.

$$37 + 2 / 5 * 68 - 2 * -$$



The next item can be considered.





$$37 + 2 / 5 * 68 - 2 * -$$



The multiplication is an operator so two items need to be popped off the stack.

$$37 + 2 / 5 * 68 - 2 * -$$



Op2:5

Second pop.





$$37 + 2 / 5 * 68 - 2 * -$$



Op2: 5

Op1: 5

The equation can be evaluated.

$$37 + 2 / 5 * 68 - 2 * -$$



Op2: 5

Op1: 5

5 * 5

The **result** can be stored on the **stack**.





$$37 + 2 / 5 * 68 - 2 * -$$



$$5 * 5 = 25$$

The next item is considered.

$$37 + 2 / 5 * 68 - 2 * -$$



6 is the **operand** so it is **pushed** onto the **stack**.





$$37 + 2 / 5 * 6 8 - 2 * -$$



The next item is considered.

$$37 + 2 / 5 * 6 8 - 2 * -$$

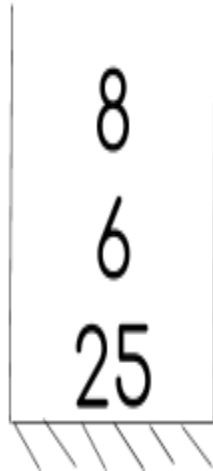


8 is the **operand** so it is **pushed** onto the **stack**.



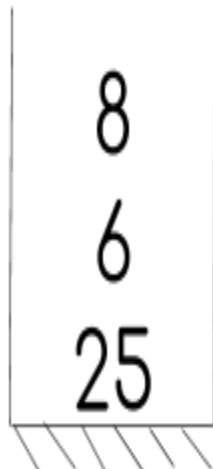


$$37 + 2 / 5 * 6 8 - 2 * -$$



The next item can be considered.

$$37 + 2 / 5 * 6 8 - 2 * -$$

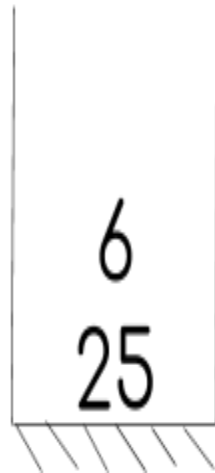


The minus sign is the **opcode**, so 2 items are **popped** from the **stack**. First pop.





$$37 + 2 / 5 * 68 - 2 * -$$



Op2: 8

Second pop.

$$37 + 2 / 5 * 68 - 2 * -$$



Op2: 8

Op1: 6

The equation can be completed.





$$37 + 2 / 5 * 68 - 2 * -$$



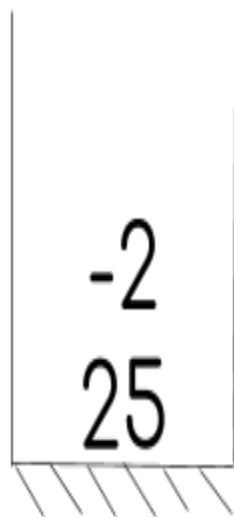
Op2: 8

Op1: 6

$$6 - 8$$

The **result** is stored on the **stack**.

$$37 + 2 / 5 * 68 - 2 * -$$



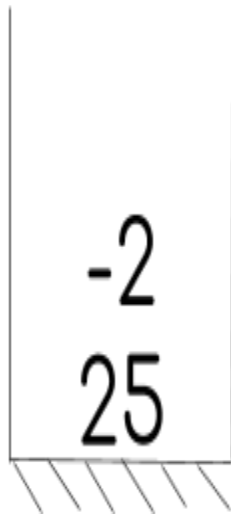
$$6 - 8 = -2$$

The next item can be considered.



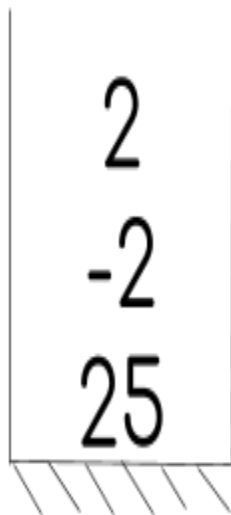


$$37 + 2 / 5 * 68 - 2 * -$$



2 is operand so it is pushed onto the stack.

$$37 + 2 / 5 * 68 - 2 * -$$

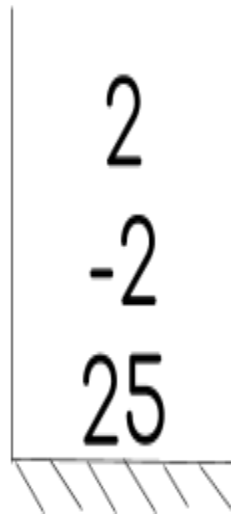


The next item is considered.



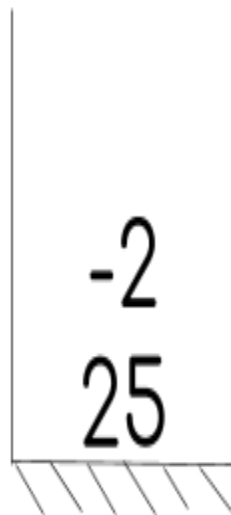


$$37 + 2 / 5 * 68 - 2 * -$$



The multiplication sign is an operator so two items are popped off the stack. First pop.

$$37 + 2 / 5 * 68 - 2 * -$$



Op2: 2

Second pop.





$$37 + 2 / 5 * 68 - 2 * -$$



Op2: 2

Op1: -2

The expression can be evaluated.

$$37 + 2 / 5 * 68 - 2 * -$$



Op2: 2

Op1: -2

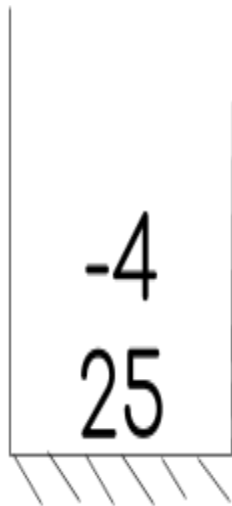
-2 * 2

The **result** is stored on the **stack**.





$$37 + 2 / 5 * 68 - 2 * -$$



$$-2 * 2 = -4$$

The next item can be considered.

$$37 + 2 / 5 * 68 - 2 * -$$



The minus sign is `opcode` so `two` items are `popped` off the `stack`. First pop.





$$37 + 2 / 5 * 68 - 2 * -$$



Op2: -4

Second pop.

$$37 + 2 / 5 * 68 - 2 * -$$



Op2: -4

Op1: 25

The expression can be derived.





$$37 + 2 / 5 * 68 - 2 * -$$



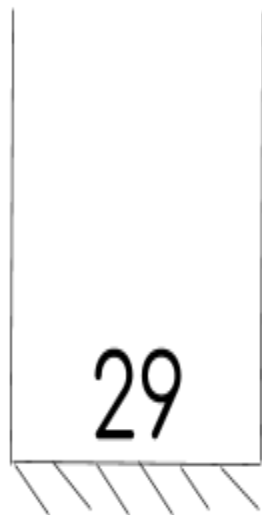
Op2: -4

Op1: 25

$$25 - -4$$

The **result** is stored on the **stack**.

$$37 + 2 / 5 * 68 - 2 * -$$



$$25 - -4 = 29$$

The next item is observed. There are no more items. The **answer** is the **top of the stack**.





$$37 + 2 / 5 * 68 - 2 * -$$



Pseudocode

Stack1 ← Stack

RPN ← Array

Op2 ← Single

Op1 ← Single

Result ← Single

For i = 0 to RPN.count - 1

 If RPN(i) = operand

 Stack1.push(RPN(i))

 ElseIf RPN(i) = opcode

 Op2 = Stack1.pop

 Op1 = Stack1.pop

 Result = Perform(RPN(i), Op1, Op2)

 Stack1.push(Result)

 End If

End For

Print Stack1.peek

Synoptic Link

Interpreters are a type of programming language translator which translates



RPN Uses

As seen above, RPN can be **executed** on a **stack**. Due to this, RPN is ideal for **interpreters** which are based on a stack, e.g. **Bytecode** and **PostScript**. For more information, follow the links listed in the extra resources section.

