Mathematical Precedence

Faced with a programming language expression such as:

```
a * b + 3 ** x / 2
```

in what order are the operations performed?

Mathematical Precedence

Faced with a programming language expression such as:

```
a * b + 3 ** x / 2
```

in what order are the operations performed?

When?	Precedence	Operations
First	Highest	`**` (exponentiation)
		`*  /  %` (multiplicative class)
Last	Lowest	`+  -` (additive class)

We are sure you remember this as "**please excuse my dear aunt sally**."

Operator Associativity

Faced with a programming language expression such as:

```
a - b - c
```

in what order are the **same-precedence** operations performed? Yes, you already know this because it has been drilled into your brain along with “Aunt Sally”: left-to-right evaluation!

```
(a - b) - c
```

Left-to-right evaluation comes from the **left-associative property** of addition and multiplicative class operations.

Why? Because the middle term `b` is *associated* with the operator *to its left*.

Right Associativity

Left-associative operators are pretty common in both mathematics and programming, what are some **right-associative** operators?

Exponentiation

$$5^{x^y} \xrightarrow{\text{in programming}} \texttt{5 ** x ** y} \xrightarrow{\text{right associative}} \texttt{5 ** (x ** y)}$$

The **middle term** is associated with the operator **on its right**.

Right Associativity

Left-associative operators are pretty common in both mathematics and programming, what are some **right-associative** operators?

Assignment (=) in C, C++, Python, Java, . . .

The following two code samples are equivilent, due to the right-associativity of value assignment

```
a = c = 3
```

```
c = 3
a = c
```

The **middle term** is associated with the operator **on its right**.

The Shunting Yard Algorithm (Ex 1)

`3 + a + b ** 5 ** q`

We begin with an empty `valueStack` and `opStack`, and traverse the elements of an expression from left to right.

	$\rightarrow$ Top of Stacks $\rightarrow$	Expression
`opStack`		
`valueStack`		`3+a+b**5**q`

Begin...

The Shunting Yard Algorithm (Ex 1)

`3 + a + b ** 5 ** q`

→ Top of Stacks → Expression

opStack

valueStack +a+b**5**q

3

Push value `3` onto `valStack`

The Shunting Yard Algorithm (Ex 1)

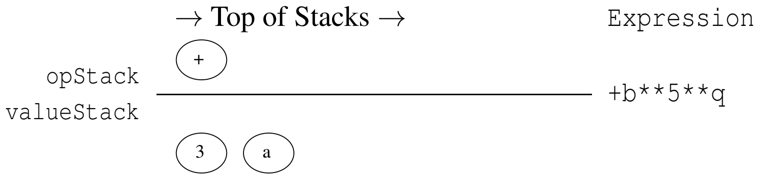
`3 + a + b ** 5 ** q`

	→ Top of Stacks →	Expression
`opStack`	+	`a+b**5**q`
`valueStack`	3	

Push operation + onto `opStack`

The Shunting Yard Algorithm (Ex 1)

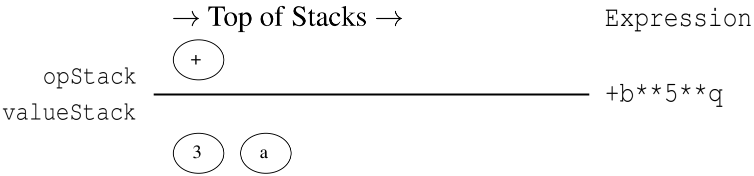
`3 + a + b ** 5 ** q`



Push value `a` onto `valStack`

The Shunting Yard Algorithm (Ex 1)

`3 + a + b ** 5 ** q`

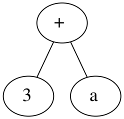


`Expression`'s **left-associative** + precedence is
$\leq$ + (`opStack`) precedence → pop the `opStack`

The Shunting Yard Algorithm (Ex 1)

`3 + a + b ** 5 ** q`

	→ Top of Stacks →	Expression
opStack		+b**5**q
valueStack		

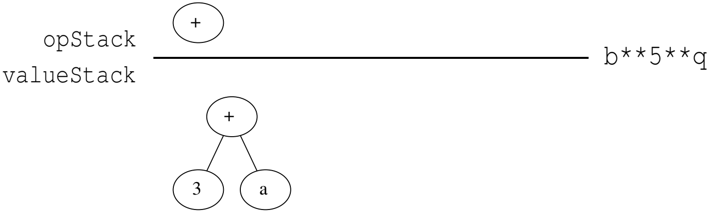


The top operation on the `opStack` binds with the top two elements of the `valueStack` the result is a new value that is pushed onto the `valueStack`.

The Shunting Yard Algorithm (Ex 1)

`3 + a + b ** 5 ** q`

→ Top of Stacks → `Expression`



Push operation + onto `opStack`

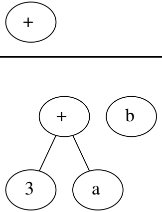
The Shunting Yard Algorithm (Ex 1)

`3 + a + b ** 5 ** q`

→ Top of Stacks → `Expression`

`opStack`

`valueStack`



`**5**q`

Push value `b` onto `valStack`

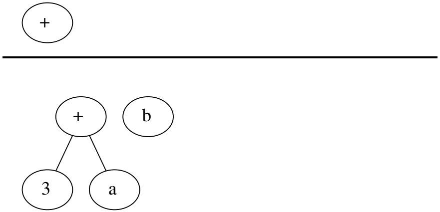
The Shunting Yard Algorithm (Ex 1)

`3 + a + b ** 5 ** q`

$\rightarrow$ Top of Stacks $\rightarrow$ `Expression`

`opStack`

`valueStack`



`**5**q`

`Expression`'s **right-associative** `**` precedence is $\geq$ + (`opStack`) precedence

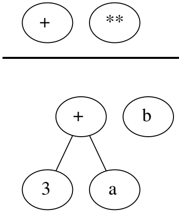
The Shunting Yard Algorithm (Ex 1)

`3 + a + b ** 5 ** q`

→ Top of Stacks → `Expression`

`opStack` `5**q`

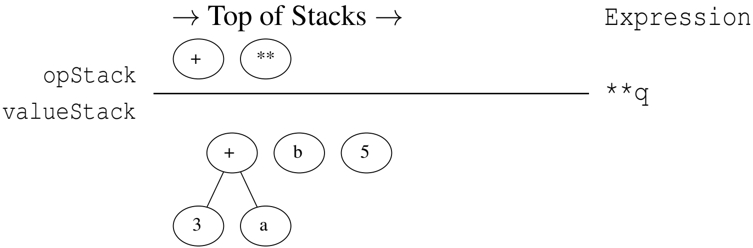
`valueStack`



Push operation ** onto `opStack`

The Shunting Yard Algorithm (Ex 1)

`3 + a + b ** 5 ** q`



Push value `5` onto `valStack`

The Shunting Yard Algorithm (Ex 1)

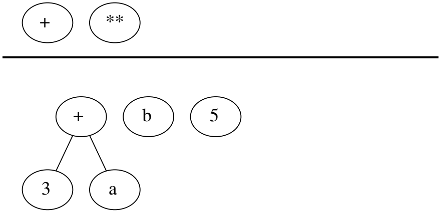
`3 + a + b ** 5 ** q`

$\rightarrow$ Top of Stacks $\rightarrow$ `Expression`

`opStack`

`valueStack`

`**q`



`Expression`'s **right-associative** `**` precedence is $\geq$ `**` (`opStack`) precedence

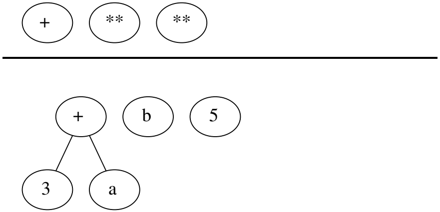
The Shunting Yard Algorithm (Ex 1)

`3 + a + b ** 5 ** q`

→ Top of Stacks → `Expression`

`opStack`

`valueStack`

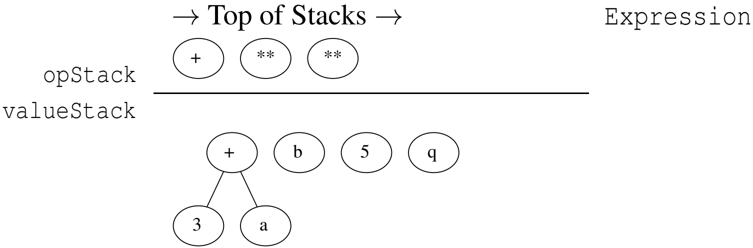


`q`

Push operation ** onto `opStack`

The Shunting Yard Algorithm (Ex 1)

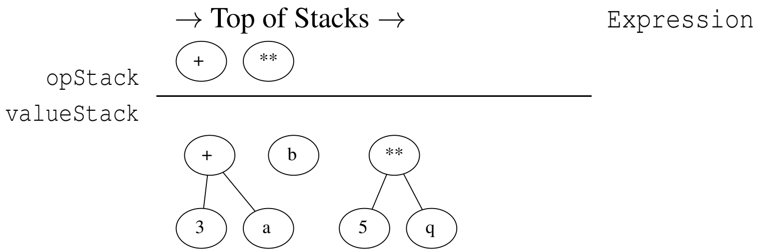
`3 + a + b ** 5 ** q`



Push value `q` onto `valStack`

The Shunting Yard Algorithm (Ex 1)

`3 + a + b ** 5 ** q`



The top operation on the `opStack` binds with the top two elements of the `valueStack` the result is a new value that is pushed onto the `valueStack`.

The Shunting Yard Algorithm (Ex 1)

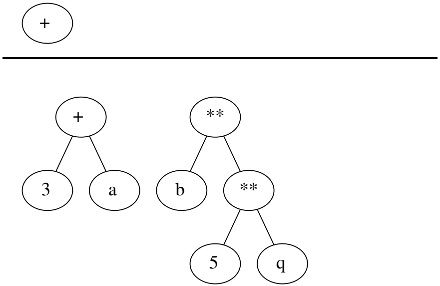
`3 + a + b ** 5 ** q`

→ Top of Stacks →

Expression

opStack

valueStack



The top operation on the `opStack` binds with the top two elements of the `valueStack` the result is a new value that is pushed onto the `valueStack`.

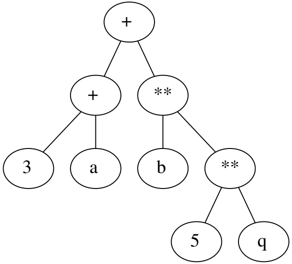
The Shunting Yard Algorithm (Ex 1)

`3 + a + b ** 5 ** q`

→ Top of Stacks → `Expression`

`opStack`

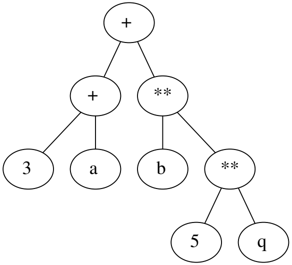
`valueStack`



The top operation on the `opStack` binds with the top two elements of the `valueStack` the result is a new value that is pushed onto the `valueStack`.

The Shunting Yard Algorithm (Ex 1)

`3 + a + b ** 5 ** q`



← Expression Tree!

Notice how right-associative exponentiation must be calculated **first** — results of higher precedence operations are the **children** of lower precedence ops.

The Shunting Yard Algorithm (Ex 2)

```
-x * a / ( b + 5 ) ** q
```

Try drawing the expression tree first . . .

can you predict the algorithm results?

The Shunting Yard Algorithm (Ex 2)

`-x * a / ( b + 5 ) ** q`

We begin with an empty `valueStack` and `opStack`, and traverse the elements of an expression from left to right.

	→ Top of Stacks →	`Expression`
`opStack`		
`valueStack`		`-x*a/(b+5)**q`

Begin. . .

The Shunting Yard Algorithm (Ex 2)

`-x * a / ( b + 5 ) ** q`

	$\rightarrow$ Top of Stacks $\rightarrow$	`Expression`
`opStack`		`*a/(b+5)**q`
`valueStack`	-x	

Push value `-x` onto `valStack`

The Shunting Yard Algorithm (Ex 2)

`-x * a / ( b + 5 ) ** q`

→ Top of Stacks → Expression

opStack *

valueStack -x

`a/(b+5)**q`

Push operation * onto `opStack`

The Shunting Yard Algorithm (Ex 2)

`-x * a / ( b + 5 ) ** q`

→ Top of Stacks → Expression

`opStack` *

`valueStack` -x a

`/(b+5)**q`

Push value `a` onto `valStack`

The Shunting Yard Algorithm (Ex 2)

`-x * a / ( b + 5 ) ** q`

$\rightarrow$ Top of Stacks $\rightarrow$ `Expression`

`opStack` *

`valueStack` -x a

`/(b+5)**q`

`Expression`'s **left-associative** / precedence is $\leq$ `*` (`opStack`) precedence $\rightarrow$ pop the `opStack`

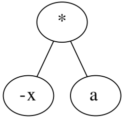
The Shunting Yard Algorithm (Ex 2)

`-x * a / ( b + 5 ) ** q`

→ Top of Stacks → Expression

opStack

valueStack `/(b+5)**q`



The top operation on the `opStack` binds with the top two elements of the `valueStack` the result is a new value that is pushed onto the `valueStack`.

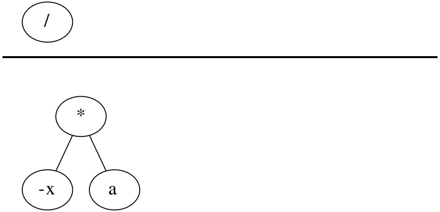
The Shunting Yard Algorithm (Ex 2)

`-x * a / ( b + 5 ) ** q`

→ Top of Stacks → `Expression`

`opStack`

`valueStack`

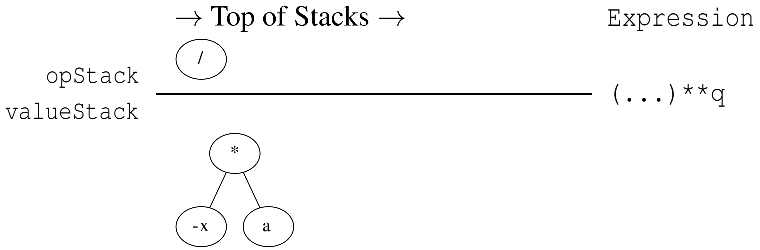


`(b+5)**q`

Push operation / onto `opStack`

The Shunting Yard Algorithm (Ex 2)

`-x * a / ( b + 5 ) ** q`



Recursive call for parenthetical grouping

The Shunting Yard Algorithm (Ex 2)

`-x * a / ( b + 5 ) ** q`

	→ Top of Stacks →	Expression
opStack		b+5
valueStack		

Begin the recursive call for subexpression tree

The Shunting Yard Algorithm (Ex 2)

`-x * a / ( b + 5 ) ** q`

$\rightarrow$ Top of Stacks $\rightarrow$ `Expression`

`opStack` `+5`

`valueStack`

b

Push value `b` onto `valStack`

The Shunting Yard Algorithm (Ex 2)

`-x * a / ( b + 5 ) ** q`

→ Top of Stacks → `Expression`

`opStack` +

`5`

`valueStack` b

Push operation + onto `opStack`

The Shunting Yard Algorithm (Ex 2)

`-x * a / ( b + 5 ) ** q`

$\rightarrow$ Top of Stacks $\rightarrow$ `Expression`

`opStack` +

`valueStack` b 5

Push value `5` onto `valStack`

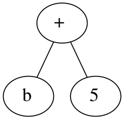
The Shunting Yard Algorithm (Ex 2)

`-x * a / ( b + 5 ) ** q`

$\rightarrow$ Top of Stacks $\rightarrow$ `Expression`

`opStack`

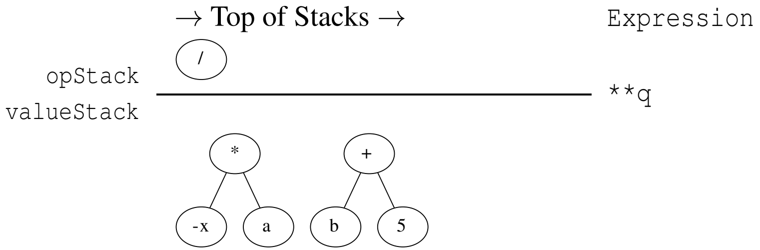
`valueStack`



The top operation on the `opStack` binds with the top two elements of the `valueStack` the result is a new value that is pushed onto the `valueStack`.

The Shunting Yard Algorithm (Ex 2)

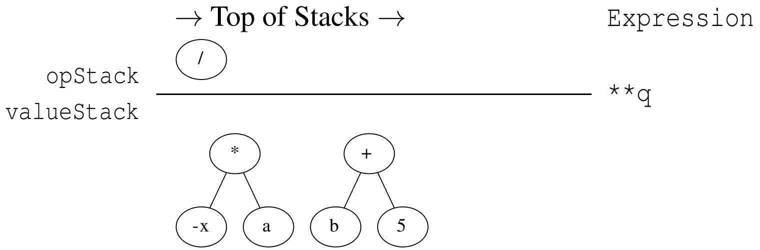
`-x * a / ( b + 5 ) ** q`



Recursive call returns — place subexpression tree on `valStack`

The Shunting Yard Algorithm (Ex 2)

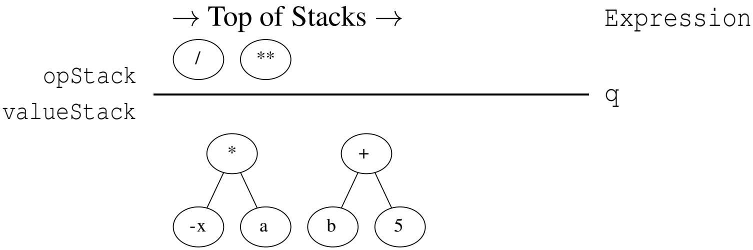
`-x * a / ( b + 5 ) ** q`



`Expression`'s **right-associative** `**` precedence is $\geq$ `/` (`opStack`) precedence

The Shunting Yard Algorithm (Ex 2)

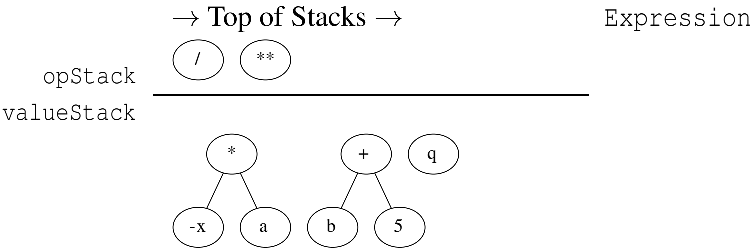
`-x * a / ( b + 5 ) ** q`



Push operation ** onto `opStack`

The Shunting Yard Algorithm (Ex 2)

`-x * a / ( b + 5 ) ** q`



Push value `q` onto `valStack`

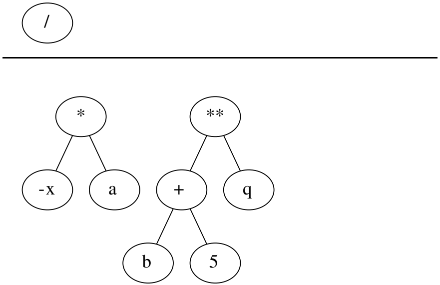
The Shunting Yard Algorithm (Ex 2)

`-x * a / ( b + 5 ) ** q`

$\rightarrow$ Top of Stacks $\rightarrow$ `Expression`

`opStack`

`valueStack`



The top operation on the `opStack` binds with the top two elements of the `valueStack` the result is a new value that is pushed onto the `valueStack`.

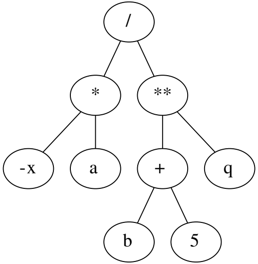
The Shunting Yard Algorithm (Ex 2)

`-x * a / ( b + 5 ) ** q`

$\rightarrow$ Top of Stacks $\rightarrow$ `Expression`

`opStack`

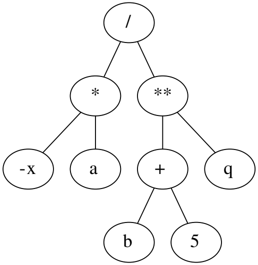
`valueStack`



The top operation on the `opStack` binds with the top two elements of the `valueStack` the result is a new value that is pushed onto the `valueStack`.

The Shunting Yard Algorithm (Ex 1)

```
-x * a / ( b + 5 ) ** q
```

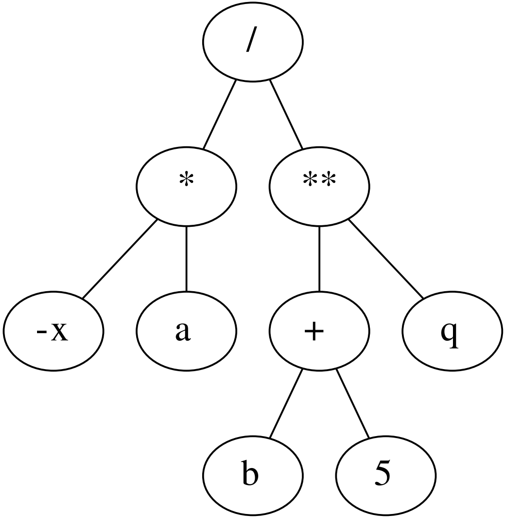


← Expression Tree!

Notice how we handled parenthetical grouping with recursion — `(...)` results are always “low” on the tree.

The Next Step . . .

```
-x * a / ( b + 5 ) ** q
```



Now that we can generate **expression trees**, how do we take advantage of the tree structure to methodically generate **machine code**?