Collections, Part One

Announcements

- Assignment 1 (Welcome to C++!) due Monday, April 15 at 2:15PM.
  - Warm up with C++!
  - Play around with strings and recursion!
- Section assignments will be announced tomorrow. If you have not signed up for a section, visit the signup link tomorrow at 5PM:

  http://cs198.stanford.edu/section
- Mac users – if you're getting an error about "minimum deployment target," we are looking into this and should get a fix posted to the course website soon. Our sincerest apologies!

Announcements

- Casual dinner for women studying CS this **Wednesday, April 10** at 5:00PM at the Gates Patio.
- Everyone is welcome!
- RSVP through link sent out last Friday, or by visiting

http://bit.ly/casualcsdinner

One last C++ detail...

Reference Parameters

- In C++, *all* parameters are passed by value unless specified otherwise.
  - The parameter is initialized to a copy of the argument.
- You can pass a parameter by reference by annotating it with the & sign:

```
void removeSpaces(string& argument);

void reverse(string& argument);
```

Yay! Now on to new things!

Organizing Data

- In order to model and solve problems, we have to have a way of representing structured data.
- We need ways of representing concepts like
  - sequences of elements,
  - sets of elements,
  - associations between elements,
  - etc.

Collections

- A **collection class** (or **container class**) is a data type used to store and organize data in some form.
- Understanding and using collection classes is critical to good software engineering.
- This week is dedicated to exploring different collections and how to harness them appropriately.
- We'll discuss efficiency issues and implementations later on.

Stack

Stack

- A **`Stack`** is a data structure representing a stack of things.
- Objects can be **pushed** on top of the stack or **popped** from the top of the stack.
- Only the top of the stack can be accessed; no other objects in the stack are visible.
- Example: Function calls

Stack

- A **`Stack`** is a data structure representing a stack of things.
- Objects can be **pushed** on top of the stack or **popped** from the top of the stack.
- Only the top of the stack can be accessed; no other objects in the stack are visible.
- Example: Function calls

Stack

- A **`Stack`** is a data structure representing a stack of things.
- Objects can be **pushed** on top of the stack or **popped** from the top of the stack.
- Only the top of the stack can be accessed; no other objects in the stack are visible.
- Example: Function calls

Stack

- A **`Stack`** is a data structure representing a stack of things.
- Objects can be **pushed** on top of the stack or **popped** from the top of the stack.
- Only the top of the stack can be accessed; no other objects in the stack are visible.
- Example: Function calls

Stack

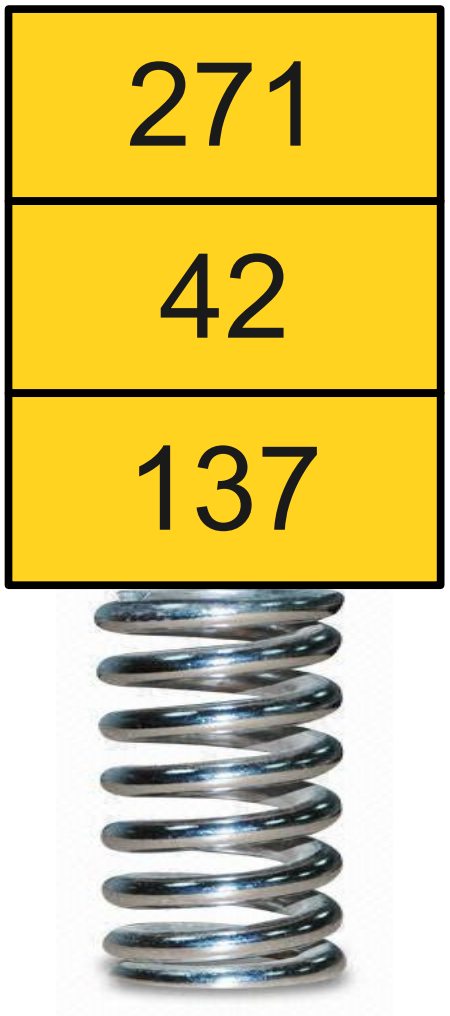
- A **`Stack`** is a data structure representing a stack of things.
- Objects can be **pushed** on top of the stack or **popped** from the top of the stack.
- Only the top of the stack can be accessed; no other objects in the stack are visible.
- Example: Function calls

Stack

- A **`Stack`** is a data structure representing a stack of things.
- Objects can be **pushed** on top of the stack or **popped** from the top of the stack.
- Only the top of the stack can be accessed; no other objects in the stack are visible.
- Example: Function calls

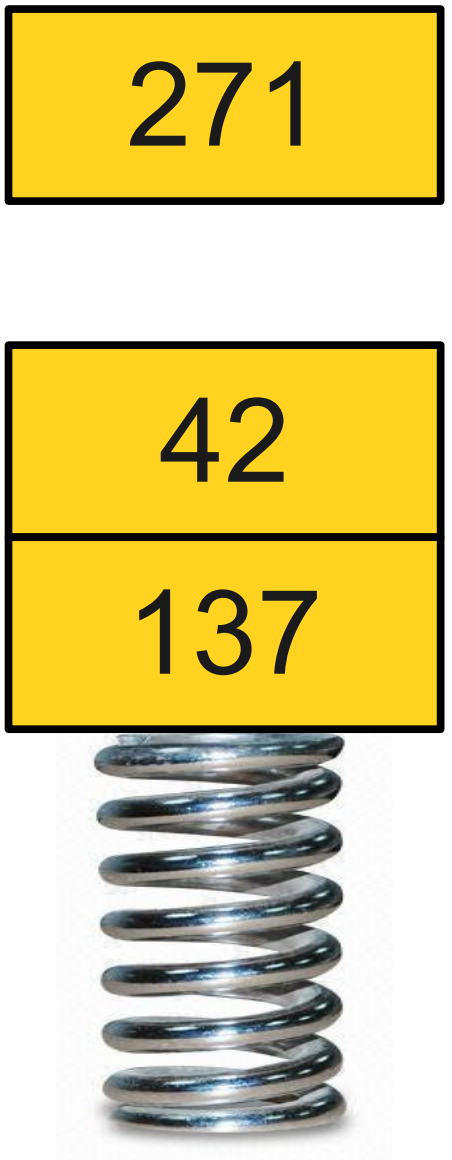
Stack

- A **`Stack`** is a data structure representing a stack of things.
- Objects can be **pushed** on top of the stack or **popped** from the top of the stack.
- Only the top of the stack can be accessed; no other objects in the stack are visible.
- Example: Function calls



Stack

- A **`Stack`** is a data structure representing a stack of things.
- Objects can be **pushed** on top of the stack or **popped** from the top of the stack.
- Only the top of the stack can be accessed; no other objects in the stack are visible.
- Example: Function calls



Stack

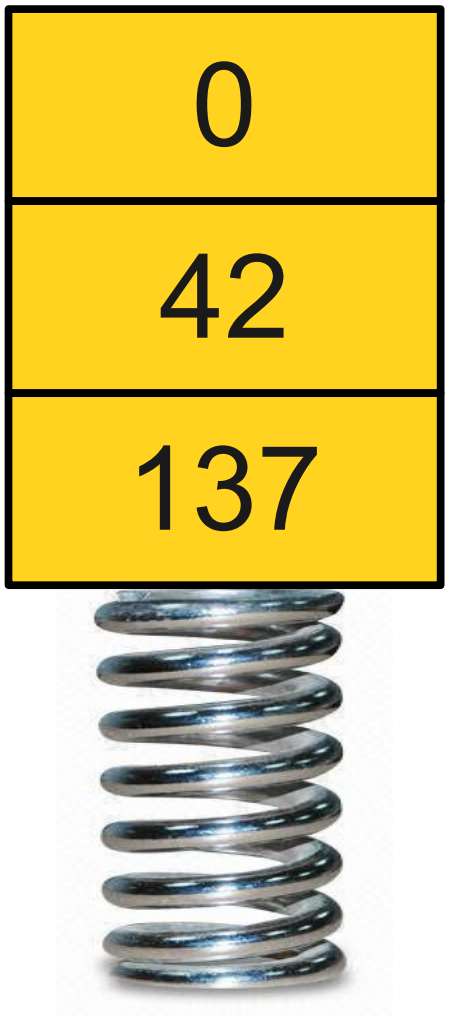
- A **`Stack`** is a data structure representing a stack of things.
- Objects can be **pushed** on top of the stack or **popped** from the top of the stack.
- Only the top of the stack can be accessed; no other objects in the stack are visible.
- Example: Function calls

Stack

- A **`Stack`** is a data structure representing a stack of things.
- Objects can be **pushed** on top of the stack or **popped** from the top of the stack.
- Only the top of the stack can be accessed; no other objects in the stack are visible.
- Example: Function calls

Stack

- A **`Stack`** is a data structure representing a stack of things.
- Objects can be **pushed** on top of the stack or **popped** from the top of the stack.
- Only the top of the stack can be accessed; no other objects in the stack are visible.
- Example: Function calls



Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
 ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
  ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
    ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
    ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
      ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
      ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
        ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
        ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
          ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
          ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
          ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
          ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
          ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
          ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
            ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
            ^
```

Balancing Parentheses

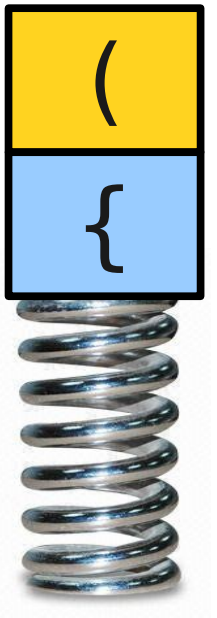
```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
            ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
          ^
```

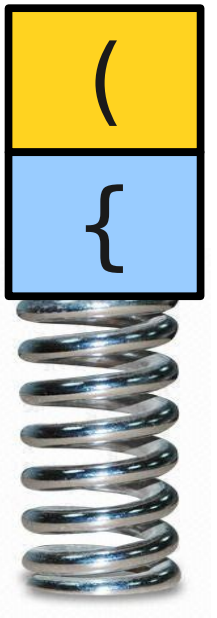
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
          ^
```



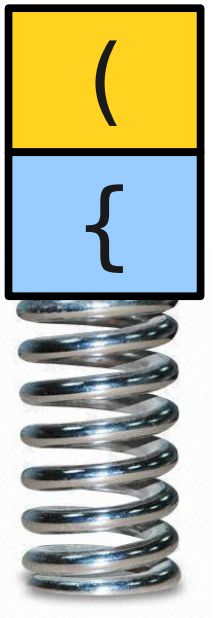
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                ^
```



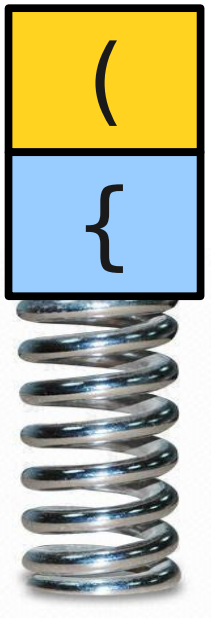
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                ^
```



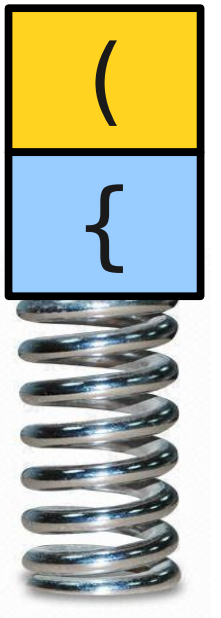
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                  ^
```



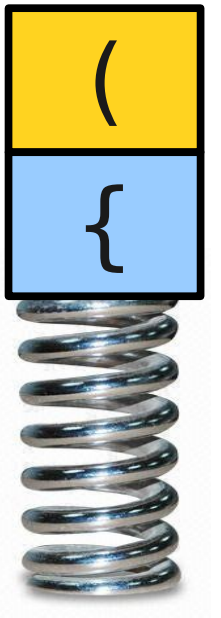
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                  ^
```



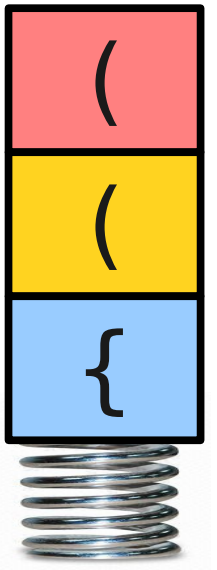
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                    ^
```



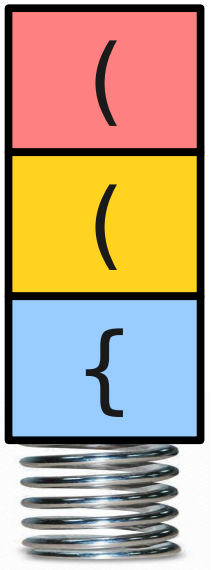
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                    ^
```



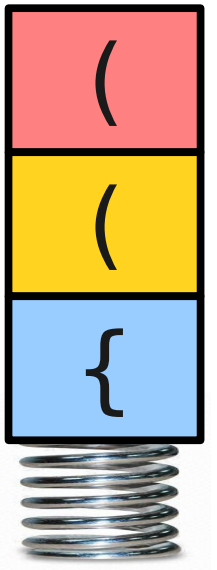
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                    ^
```



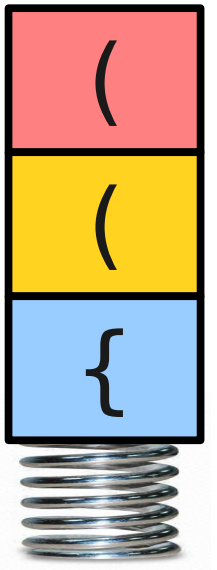
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                    ^
```



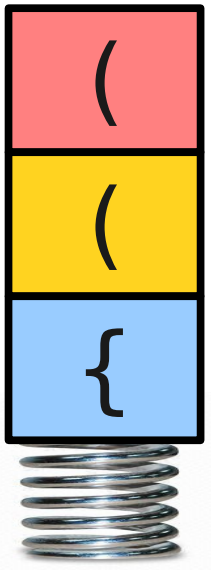
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                        ^
```



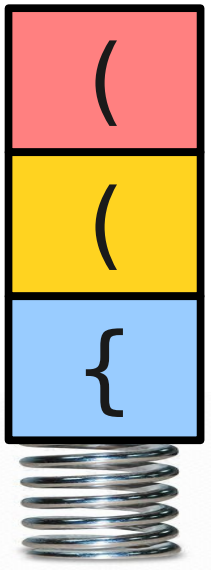
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                          ^
```



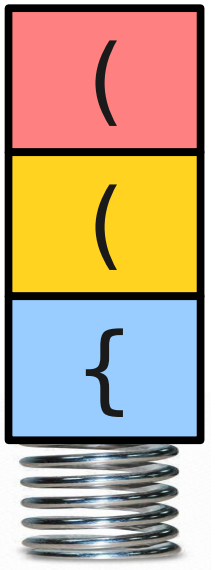
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                          ^
```



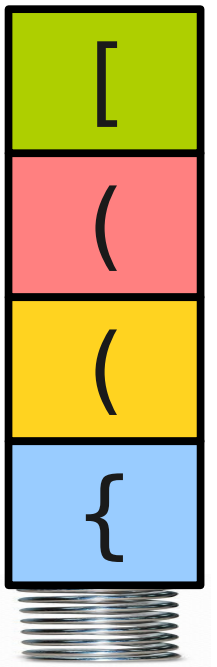
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                          ^
```



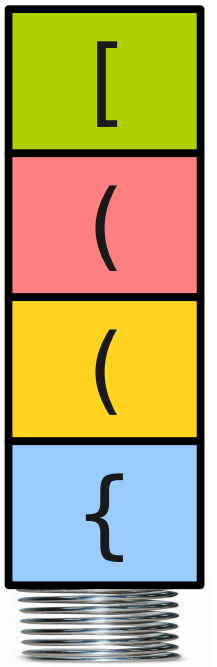
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                          ^
```



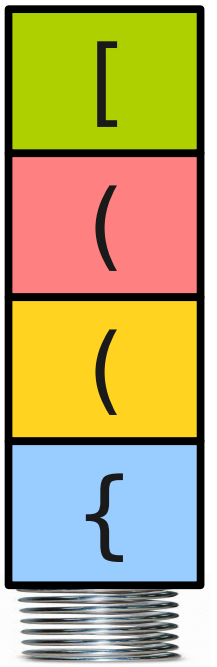
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                          ^
```



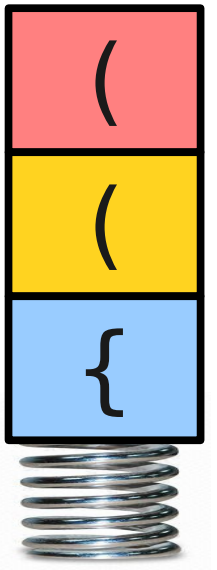
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                          ^
```



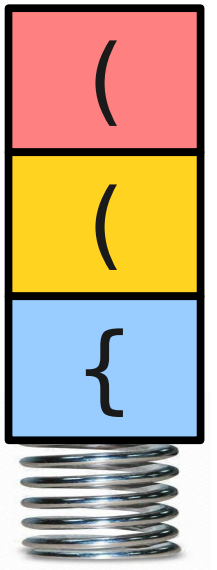
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                           ^
```



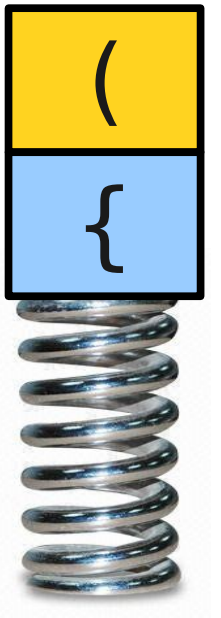
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                              ^
```



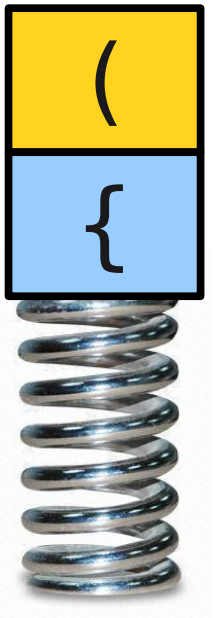
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                ^
```



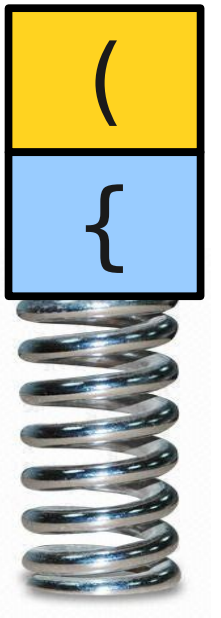
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                ^
```



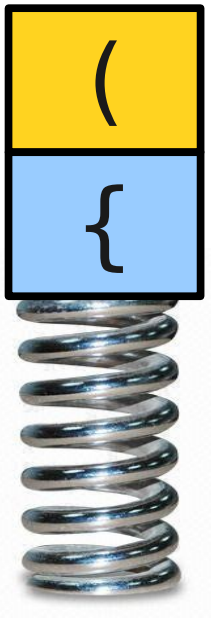
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                ^
```



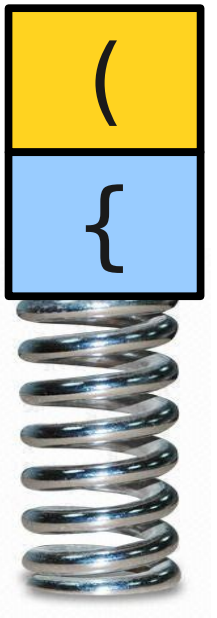
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                 ^
```



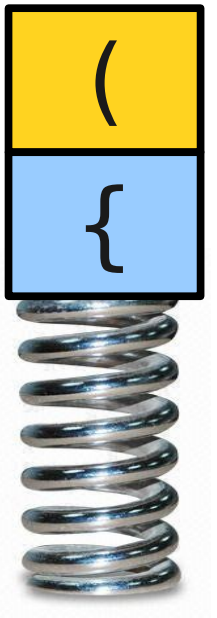
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                  ^
```



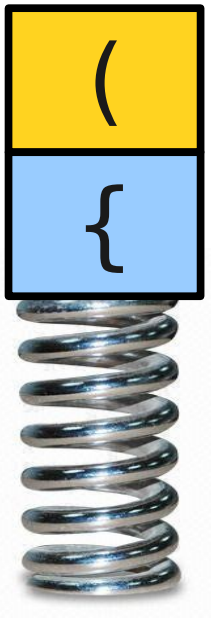
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                  ^
```



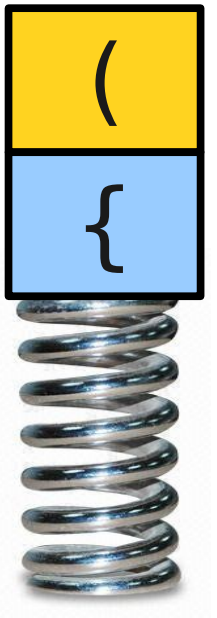
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                  ^
```



Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                  ^
```



Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                  ^
```

Balancing Parentheses

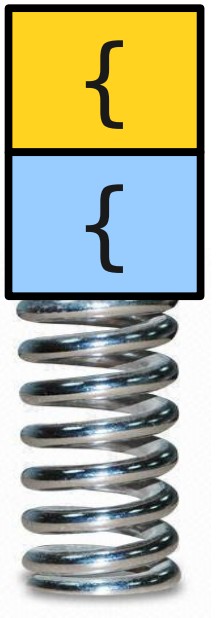
```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                      ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                      ^
```

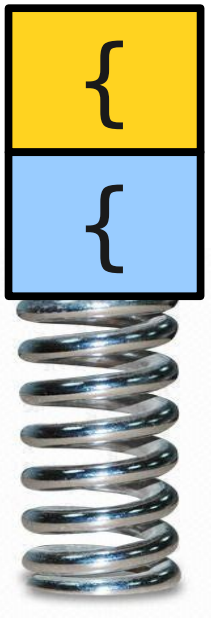
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                      ^
```



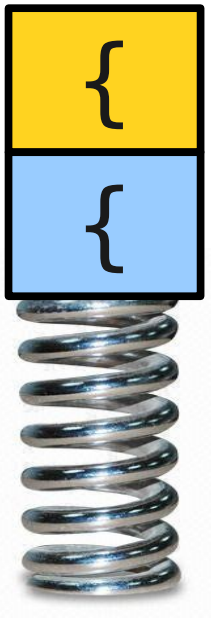
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                      ^
```



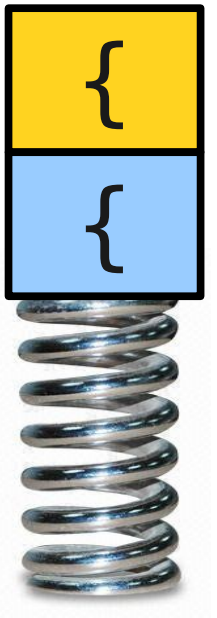
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                        ^
```



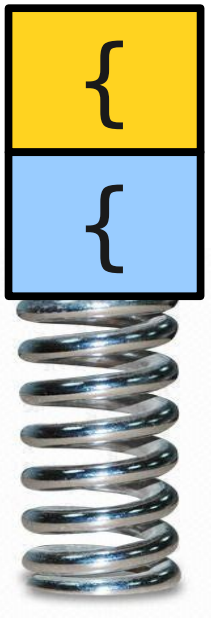
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                      ^
```



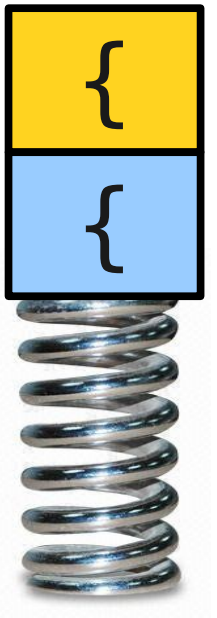
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                        ^
```



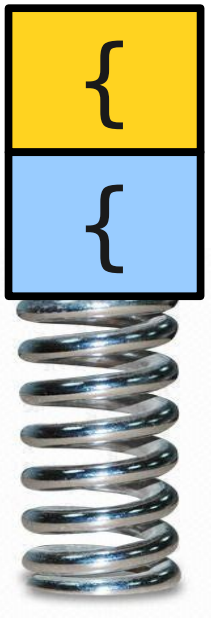
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                          ^
```



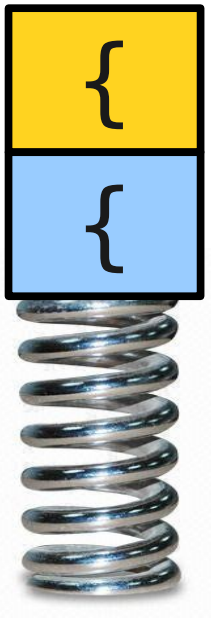
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                          ^
```



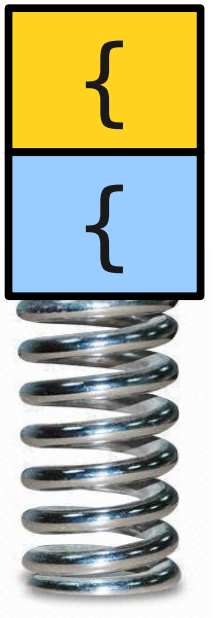
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                          ^
```



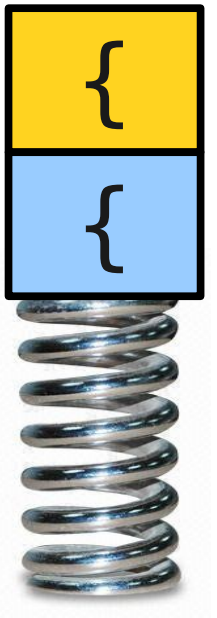
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                              ^
```



Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                            ^
```



Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                              ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                                ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                                ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                                ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                                 ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```

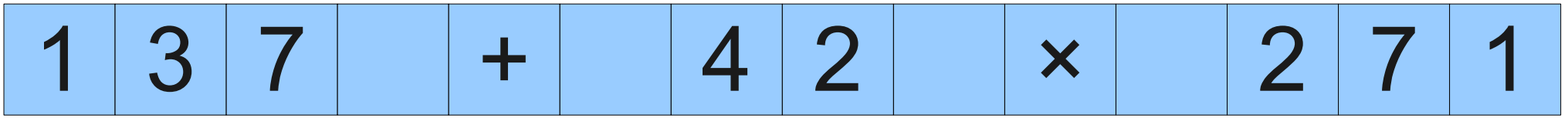
Interesting exercise: code this up!

Application: Evaluating Expressions

Evaluating Expressions

- Evaluating expressions is much trickier than it might seem due to issues of precedence.
  - 1 + 3 * 5 – 7 = 9
  - 4 / 2 + 2 = 4
  - 17 % 6 % 3 = 2
- How do we evaluate an expression?

The Challenge



Evaluating Expressions

- Two separate concerns in evaluating expressions:
  - **Scanning** the string and breaking it apart into its constituent components (*tokens*).
  - **Parsing** the tokens to determine what expression is encoded.
- For now, let's assume we have a scanner. How might we handle parsing?

The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

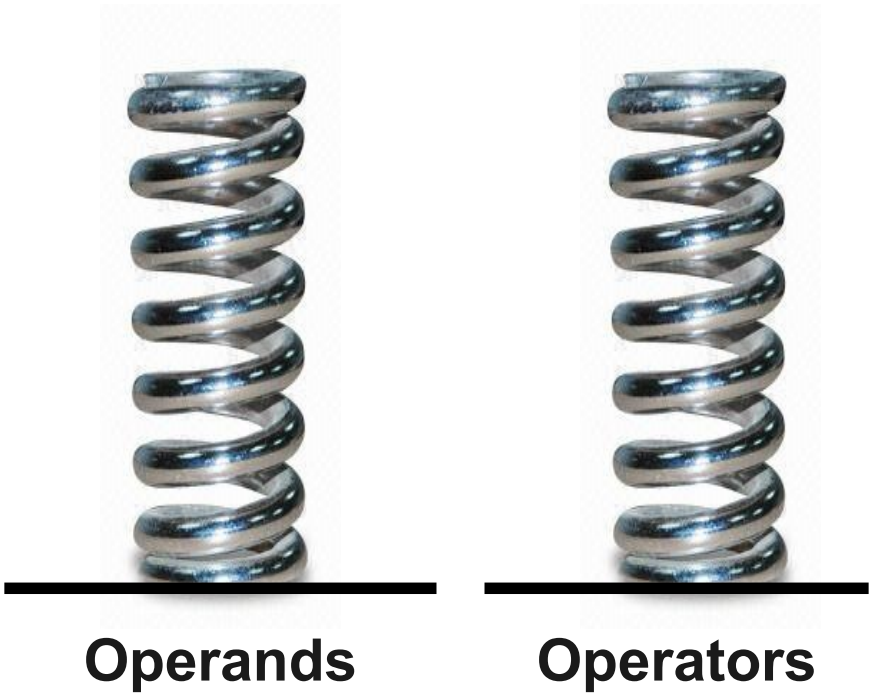
The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

Operands

The Shunting-Yard Algorithm

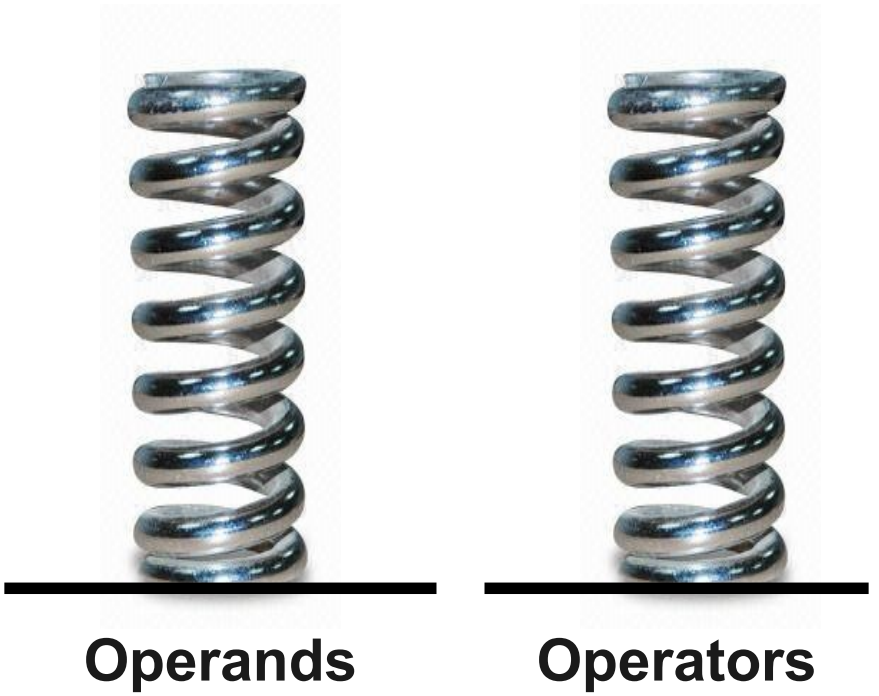
2	+	3	*	5	-	6	/	2



The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

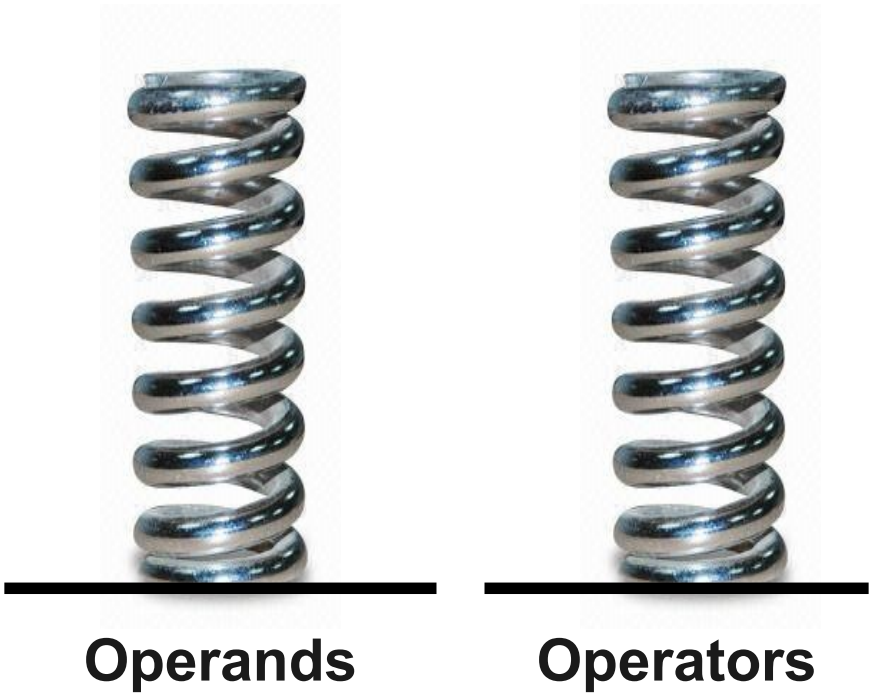
2	+	3	*	5	-	6	/	2



The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

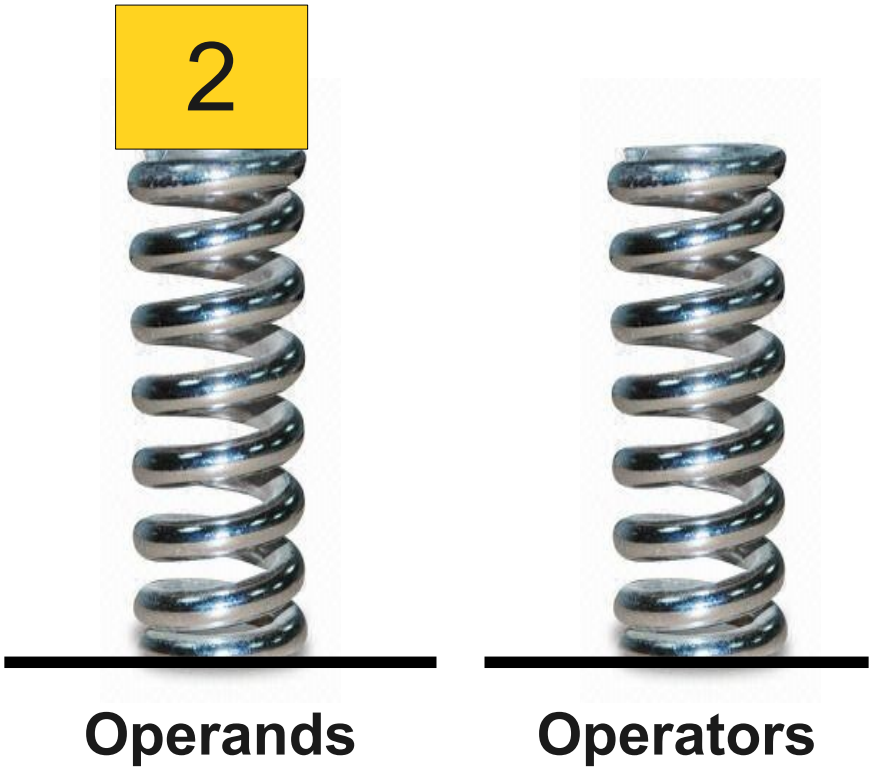
2	+	3	*	5	-	6	/	2



The Shunting-Yard Algorithm

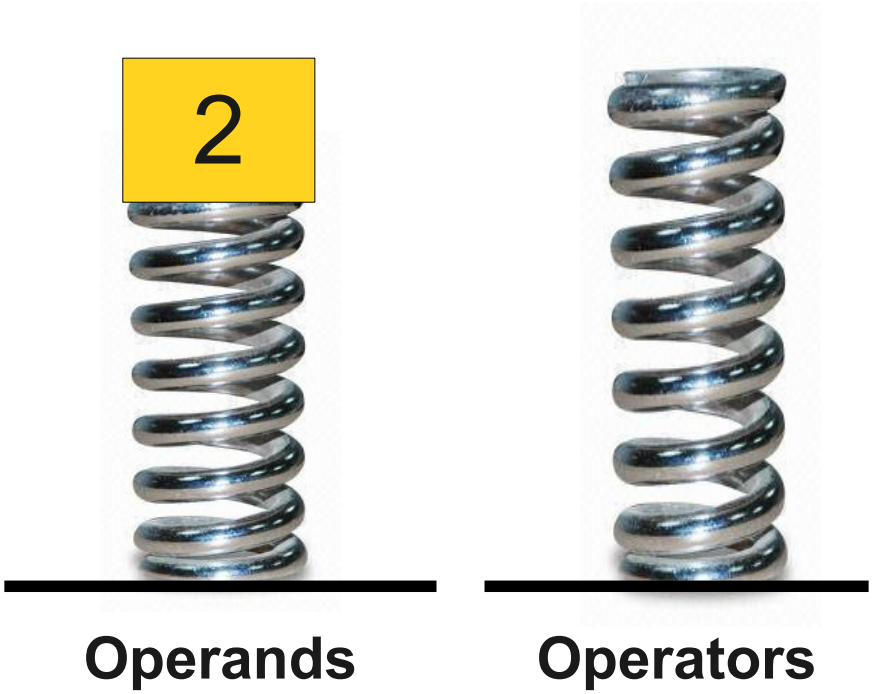
2	+	3	*	5	-	6	/	2

+	3	*	5	-	6	/	2



The Shunting-Yard Algorithm

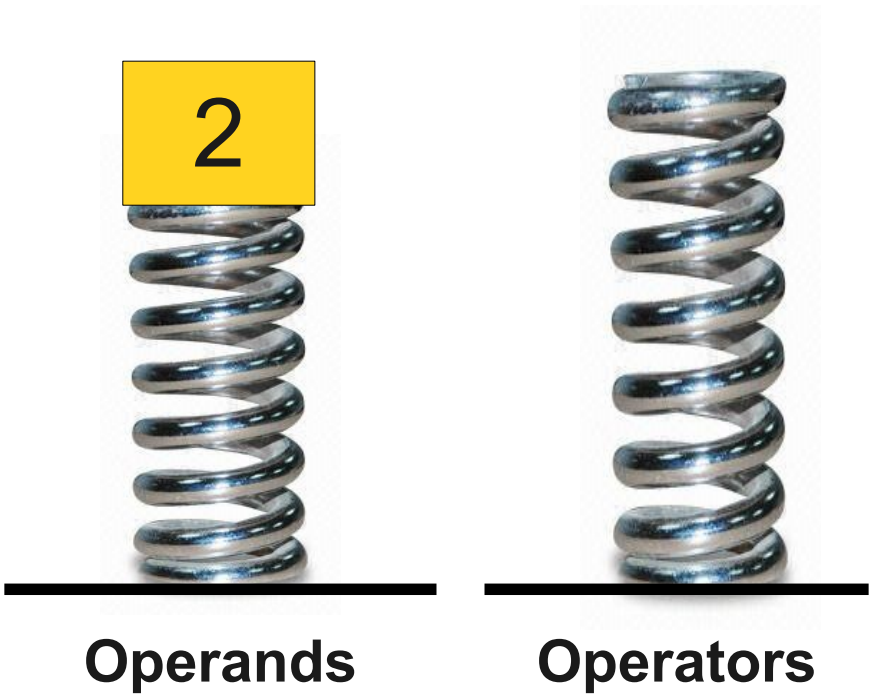
2	+	3	*	5	-	6	/	2
	+	3	*	5	-	6	/	2



The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

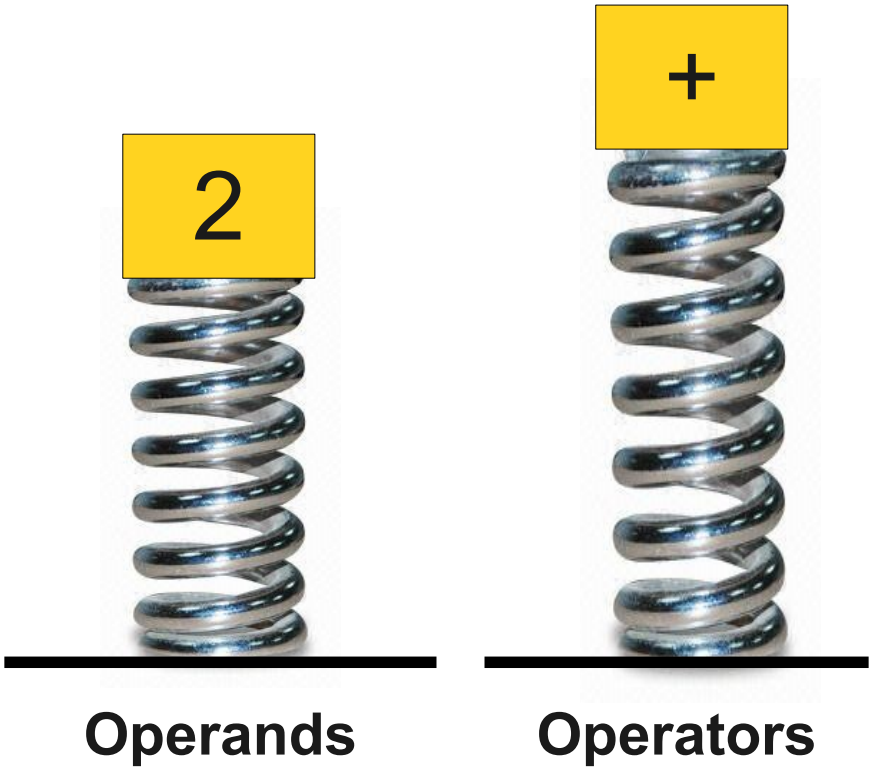
+	3	*	5	-	6	/	2



The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

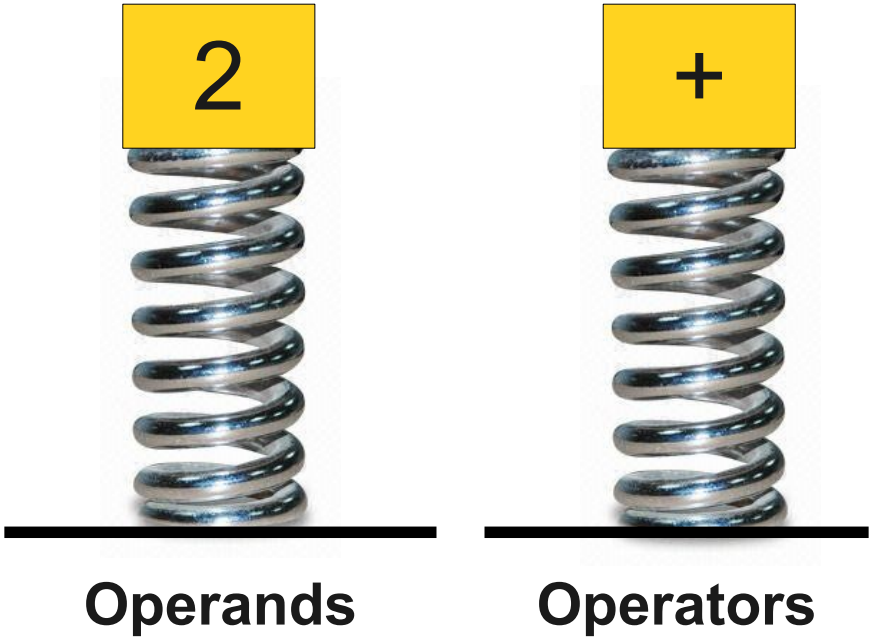
3	*	5	-	6	/	2



The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

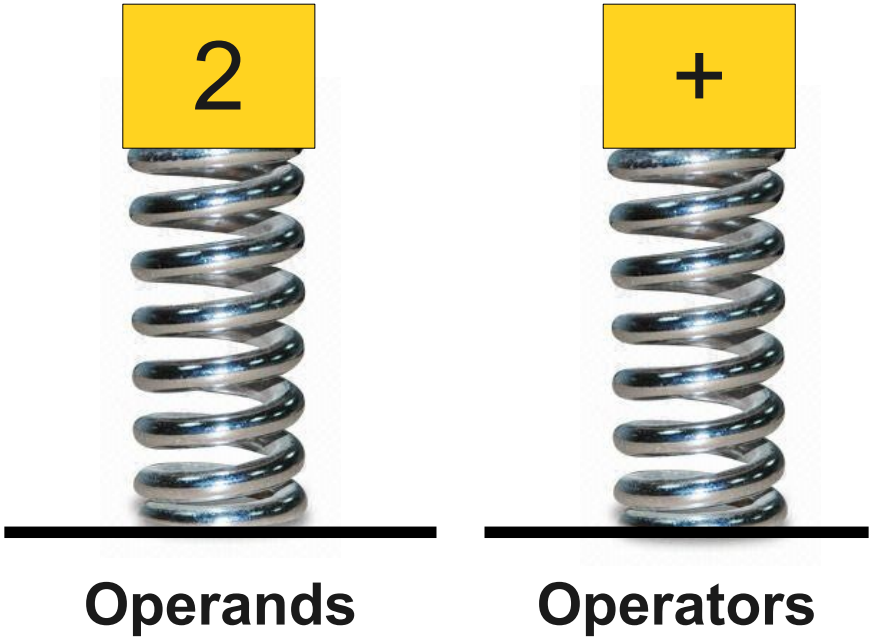
3	*	5	-	6	/	2



The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

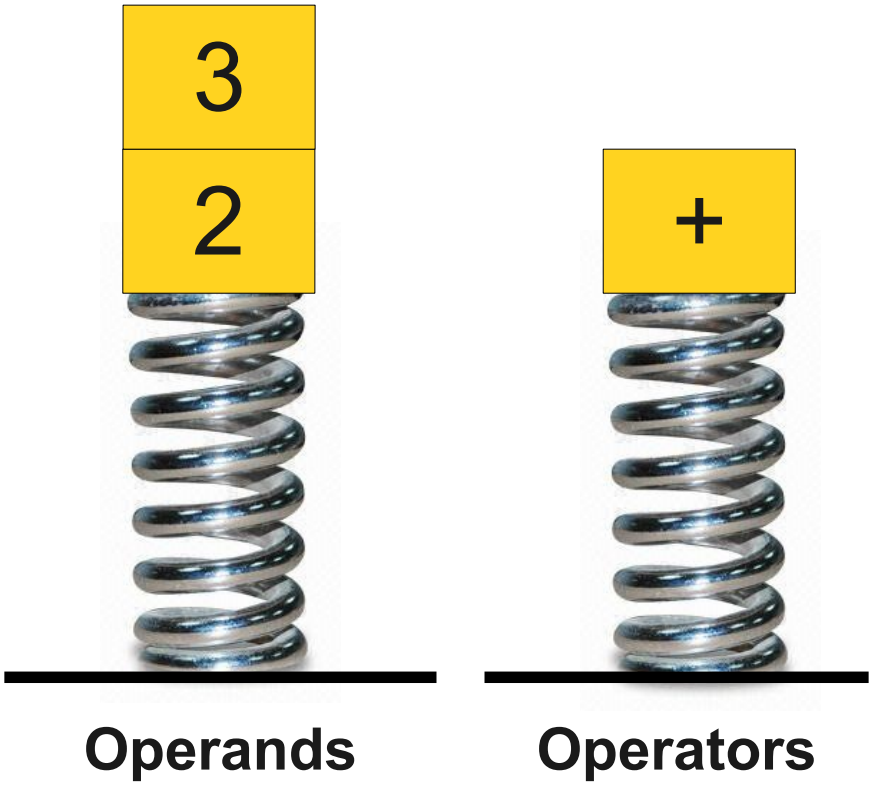
3	*	5	-	6	/	2



The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

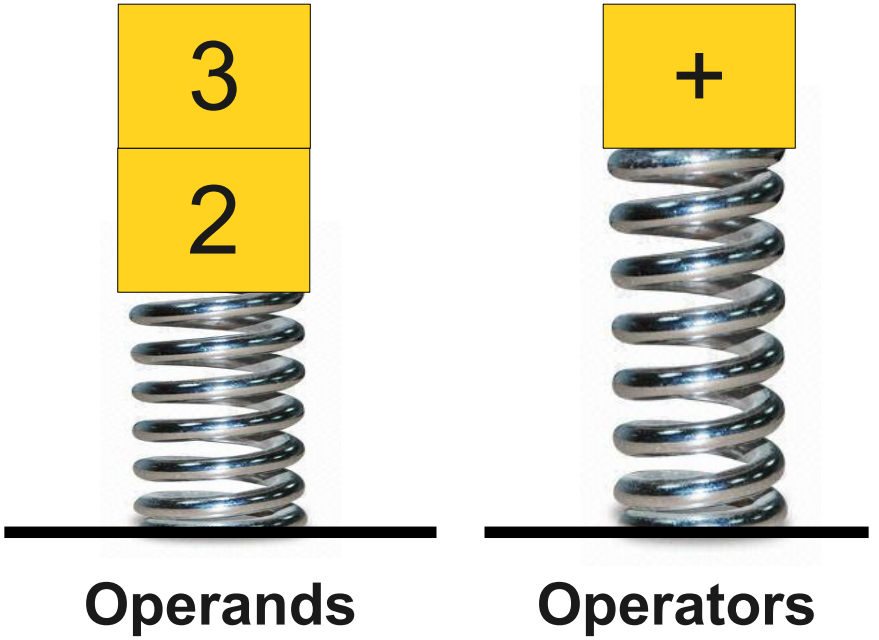
*	5	-	6	/	2



The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

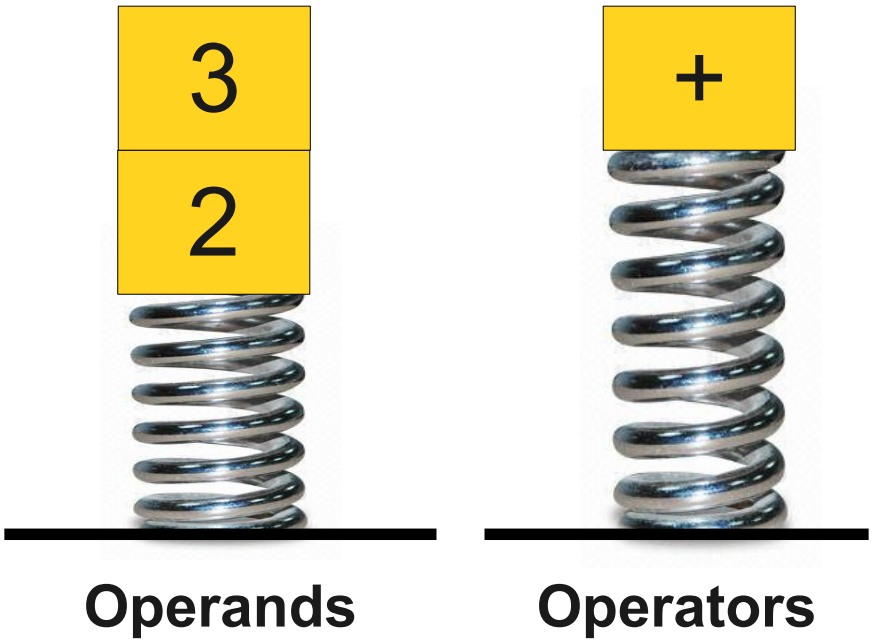
*	5	-	6	/	2



The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

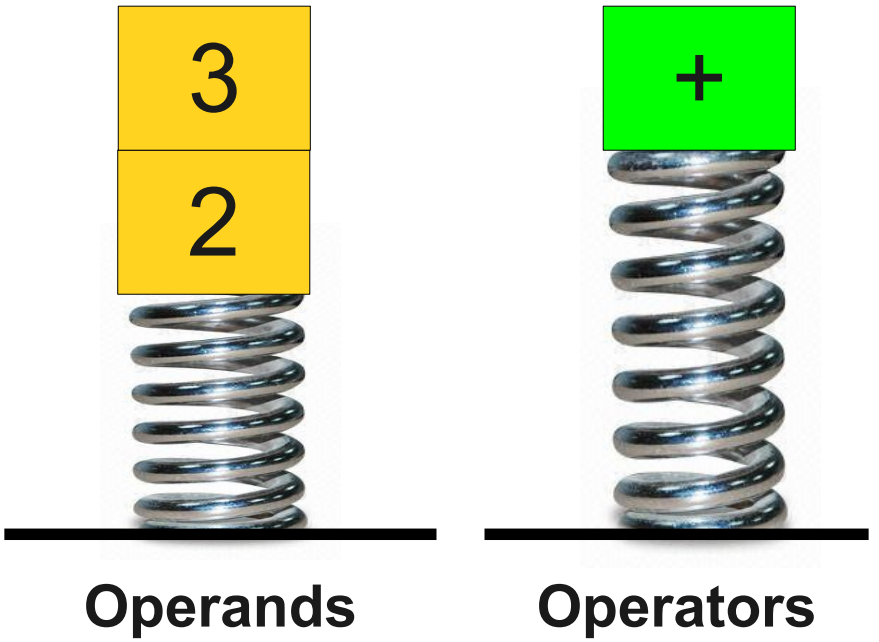
*	5	-	6	/	2



The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

*	5	-	6	/	2



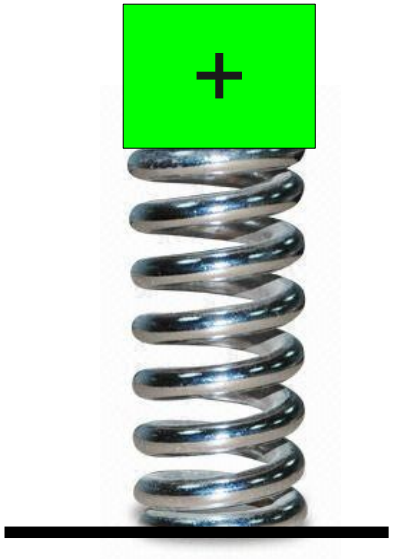
The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

*	5	-	6	/	2

Multiplication has higher precedence than addition, so we will postpone the addition until after we've done the multiplication.

Operands

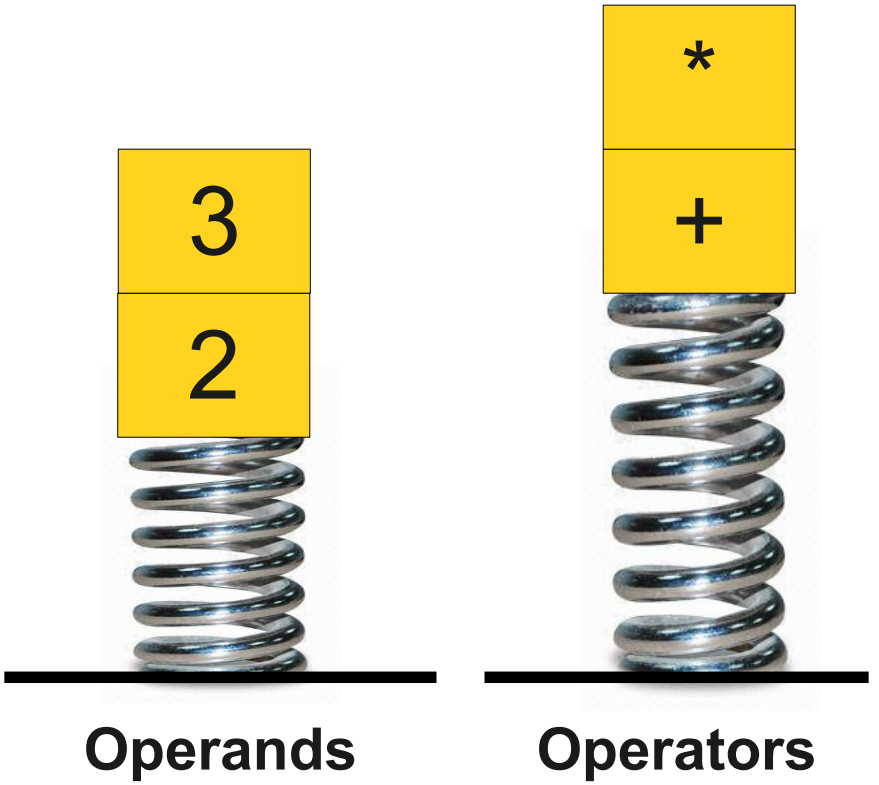


Operators

The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

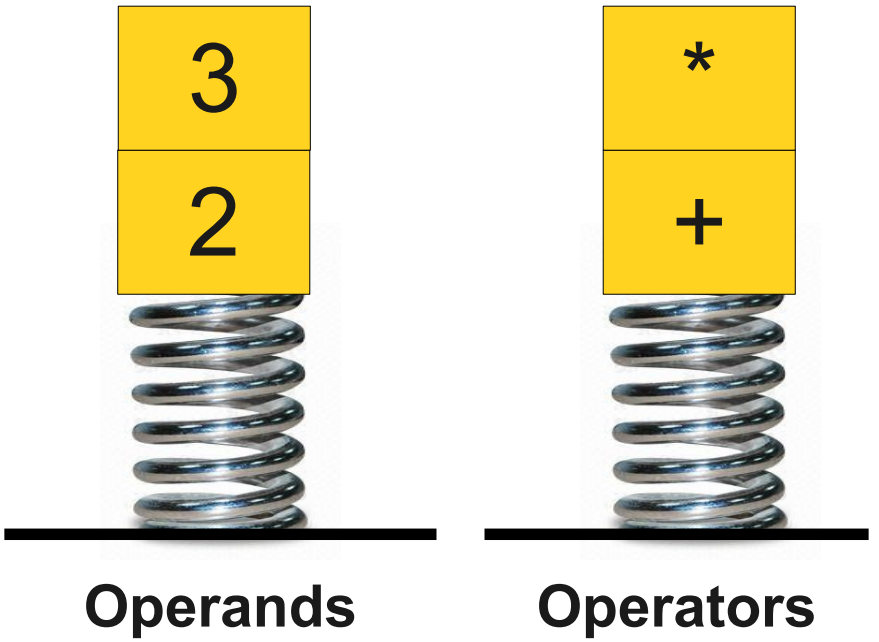
5	-	6	/	2



The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

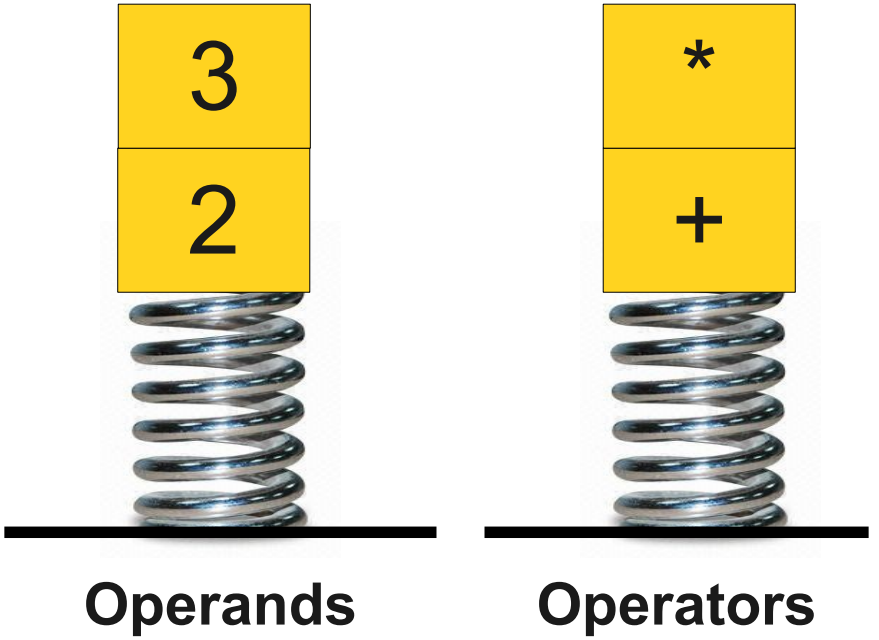
5	-	6	/	2



The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

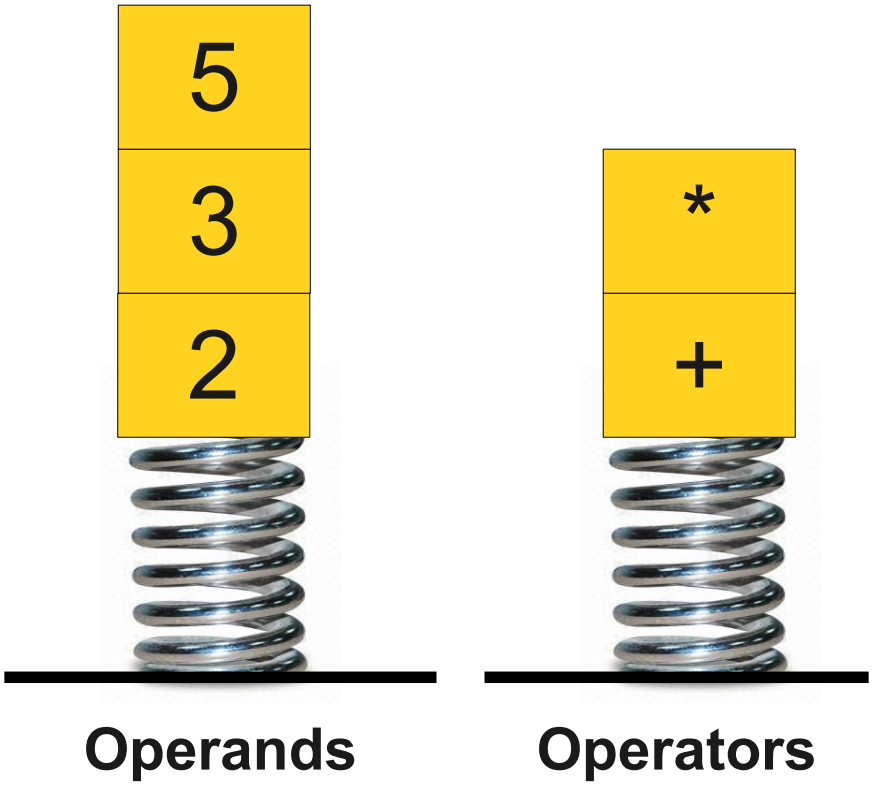
5	-	6	/	2



The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

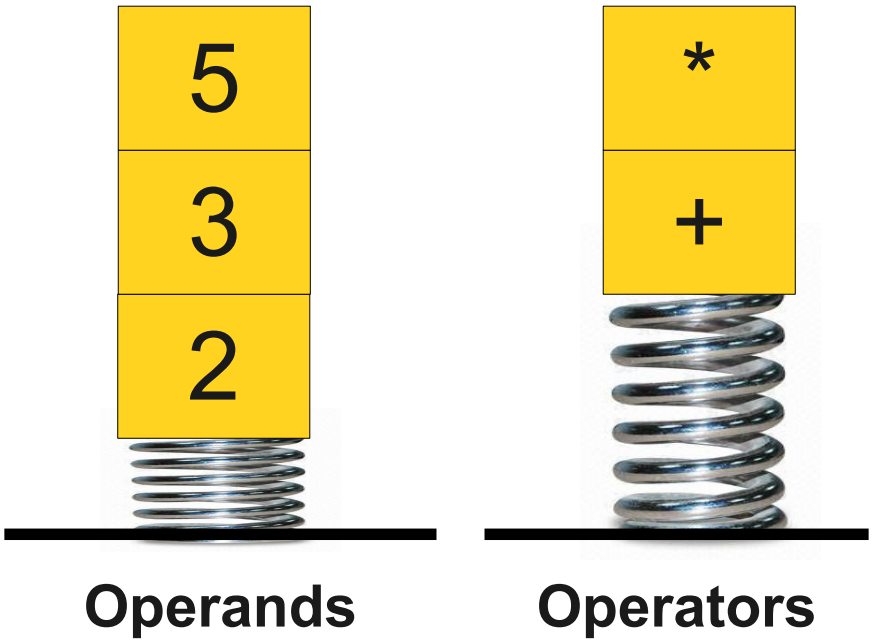
-	6	/	2



The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

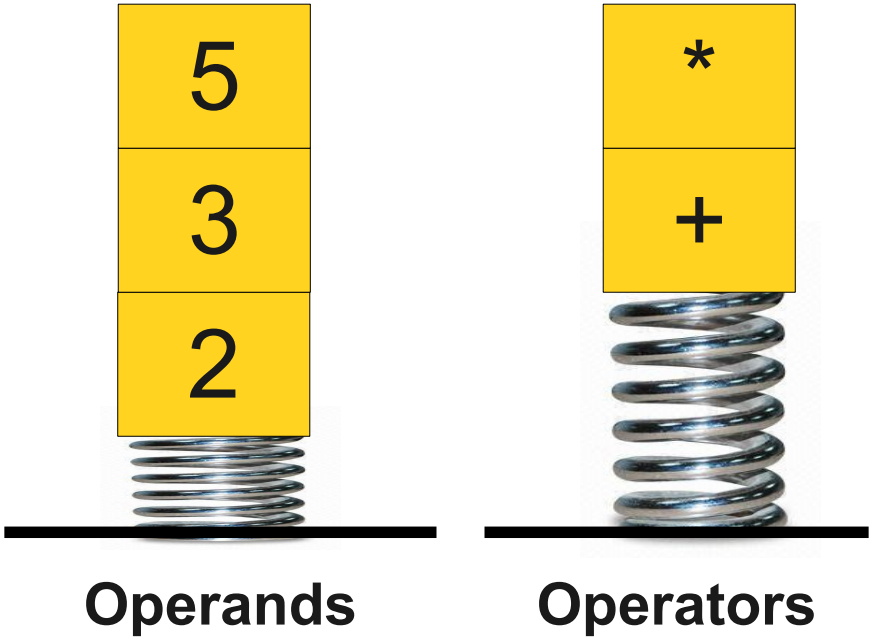
-	6	/	2



The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

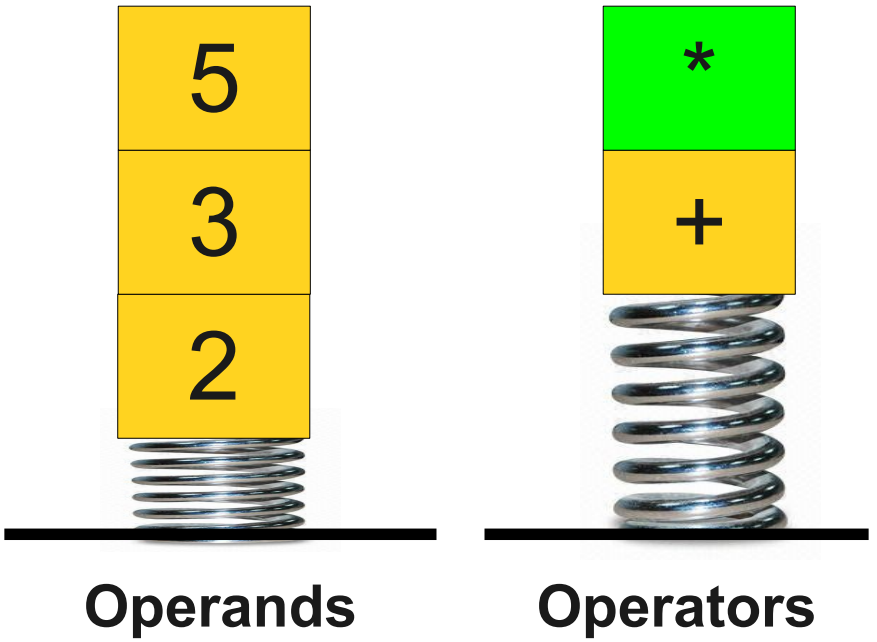
-	6	/	2



The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

-	6	/	2

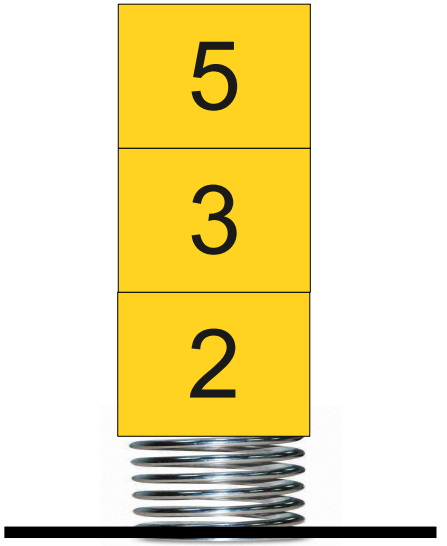


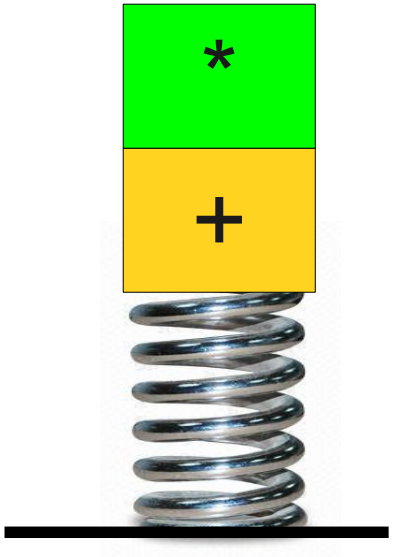
The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

-	6	/	2

Subtraction has lower precedence than multiplication, so we need to evaluate the multiply before the subtract.



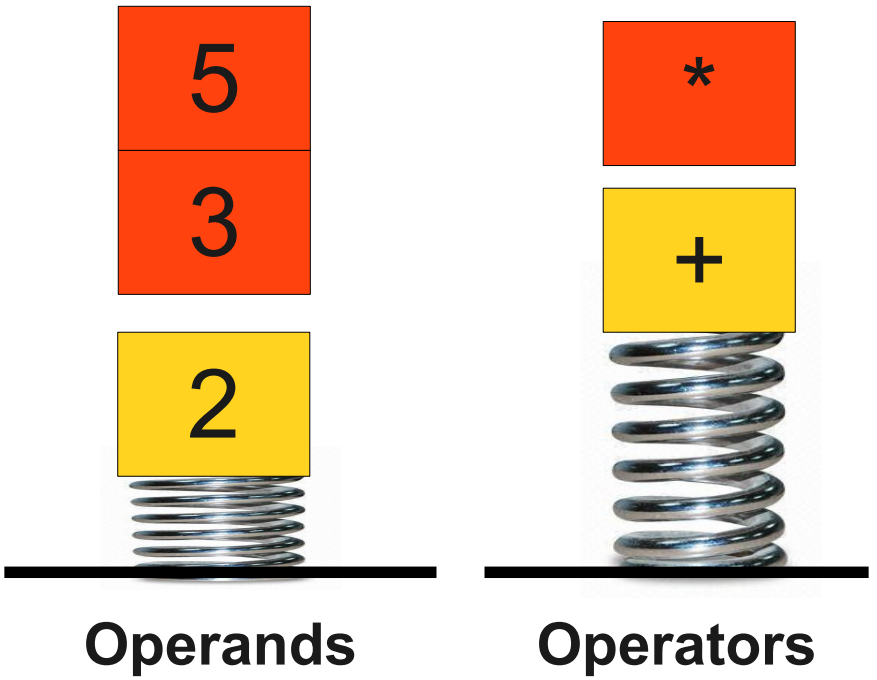


Operands **Operators**

The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

-	6	/	2

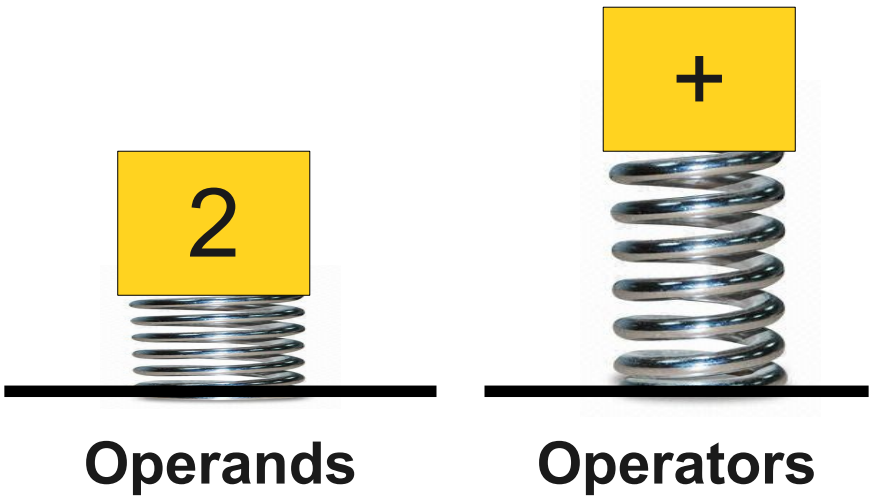


The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

-	6	/	2

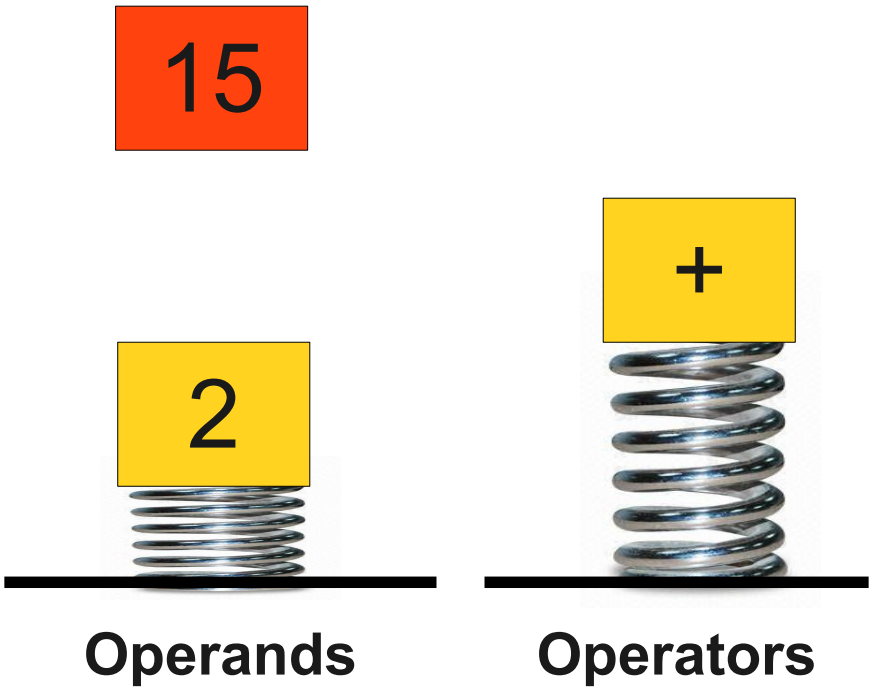
3	*	5



The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

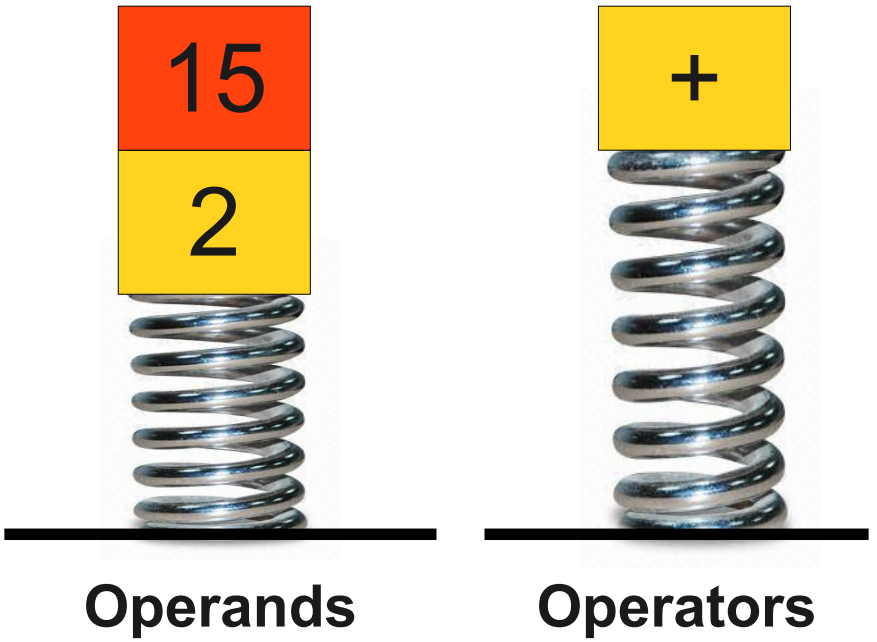
-	6	/	2



The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

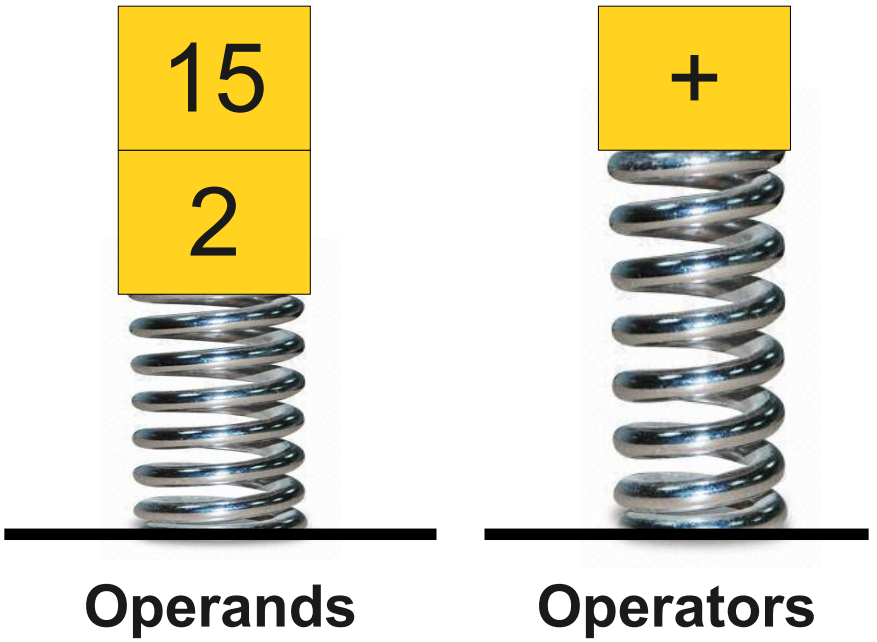
-	6	/	2



The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

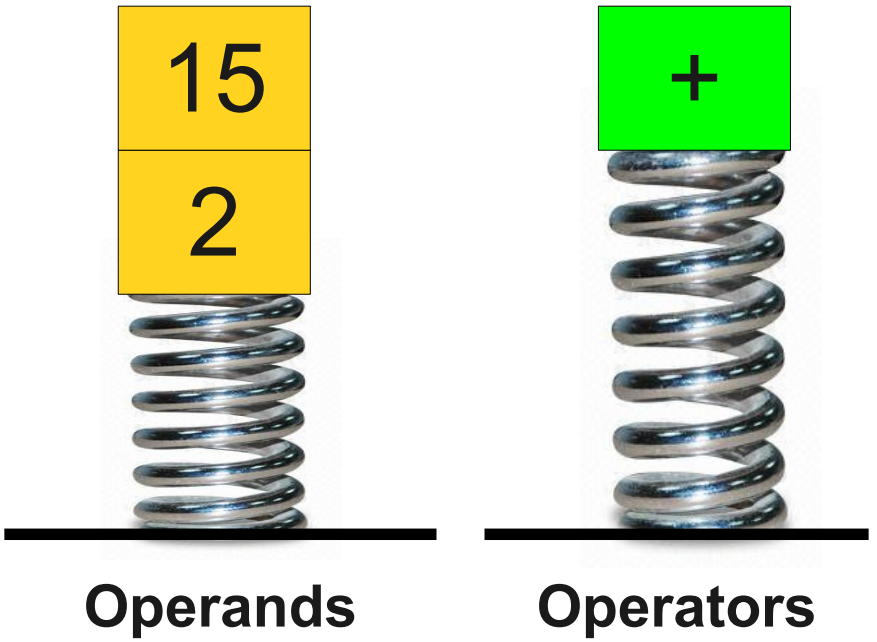
-	6	/	2



The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

-	6	/	2



The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

-	6	/	2

Subtraction has equal precedence to addition. Since addition is left-associative, we evaluate the add before the subtract.

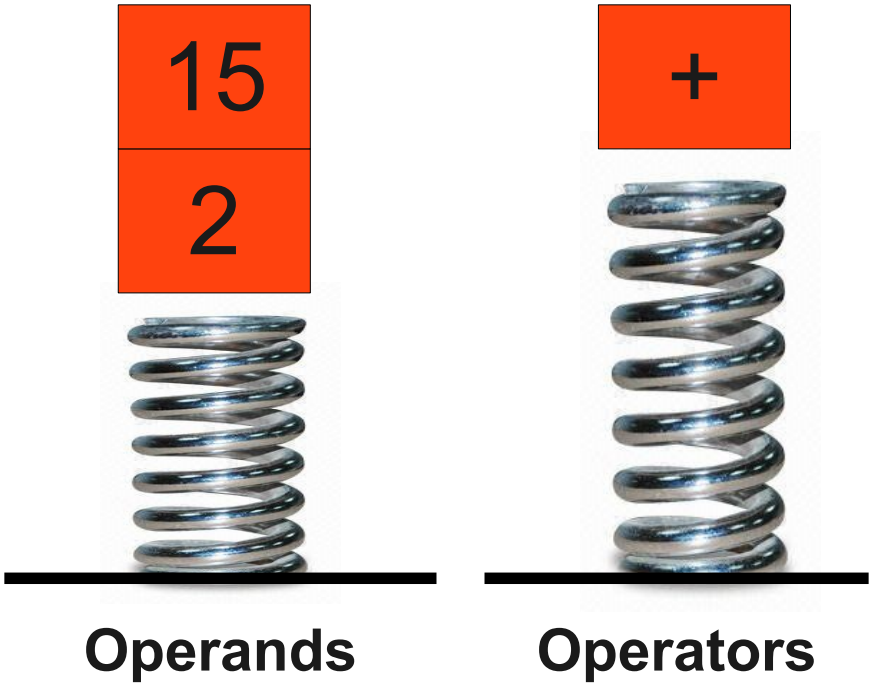
Operands

Operators

The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

-	6	/	2

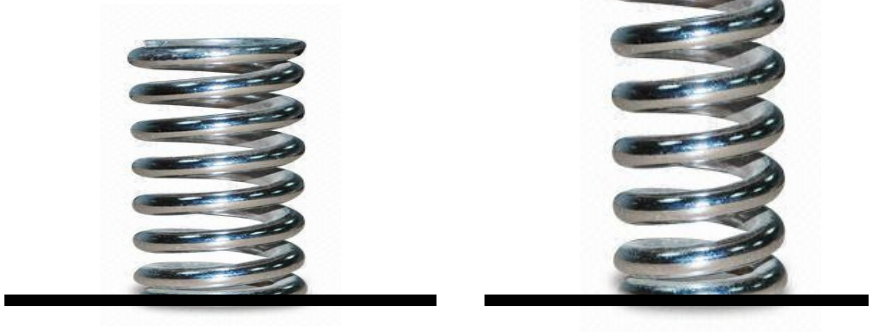


The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

-	6	/	2

2	+	15



Operands **Operators**

The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

-	6	/	2

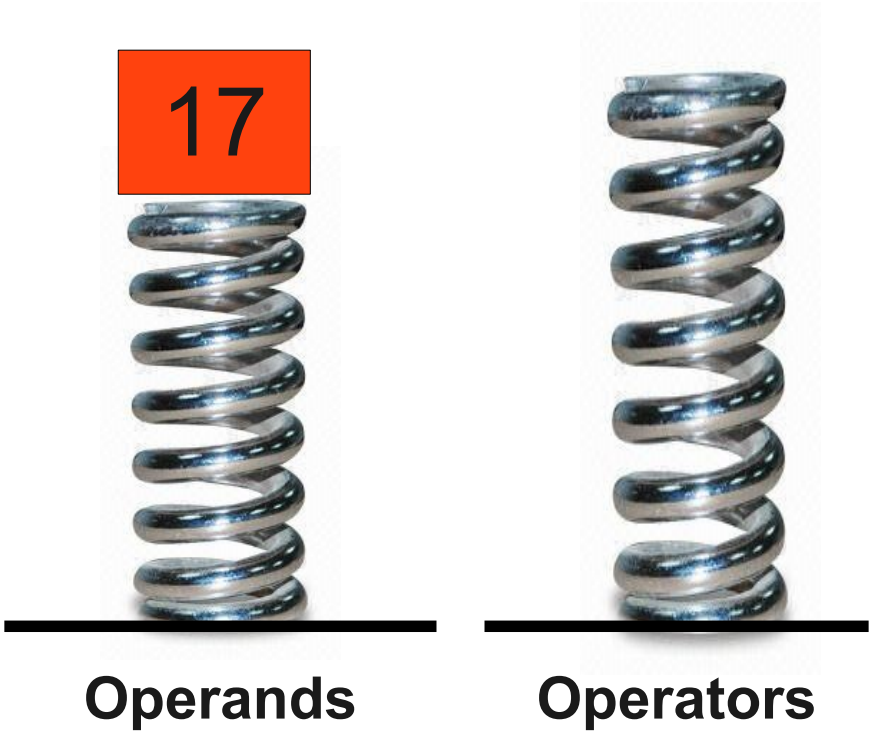
17

Operands **Operators**

The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

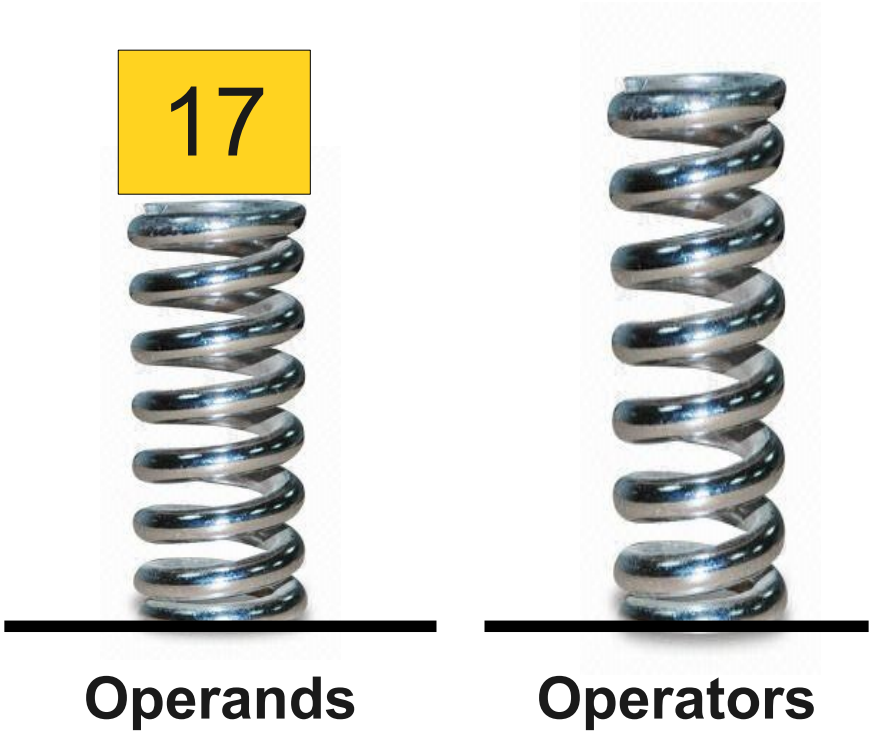
-	6	/	2



The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

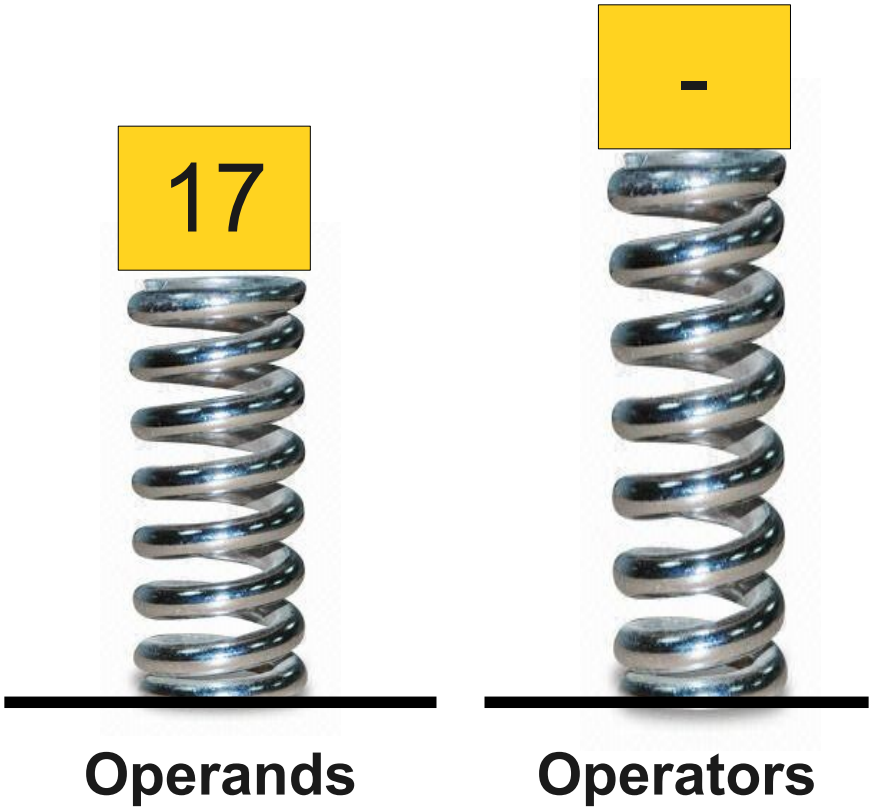
-	6	/	2



The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

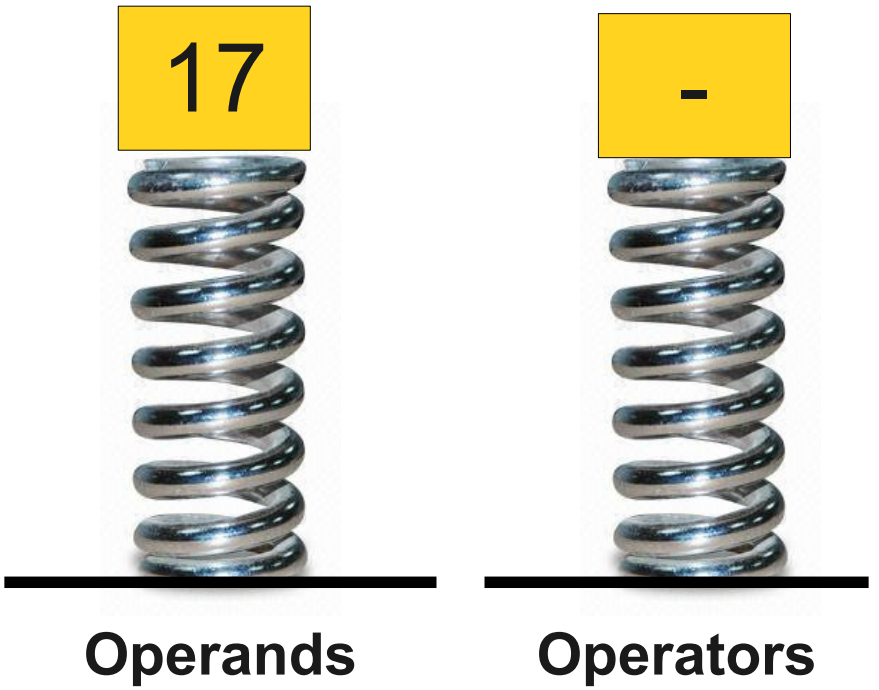
6	/	2



The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

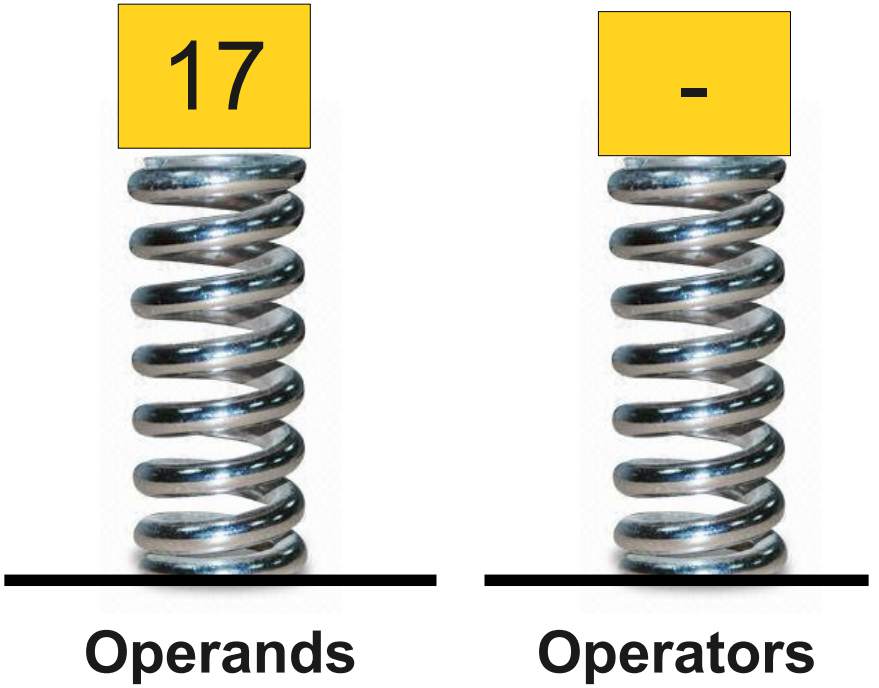
6	/	2



The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

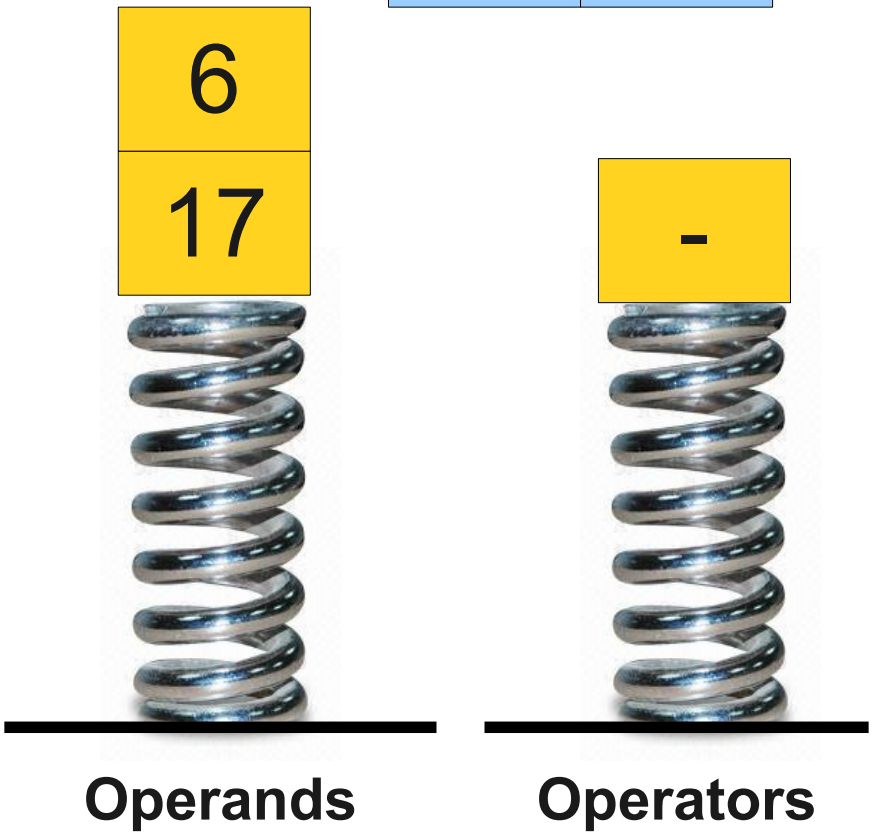
6	/	2



The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

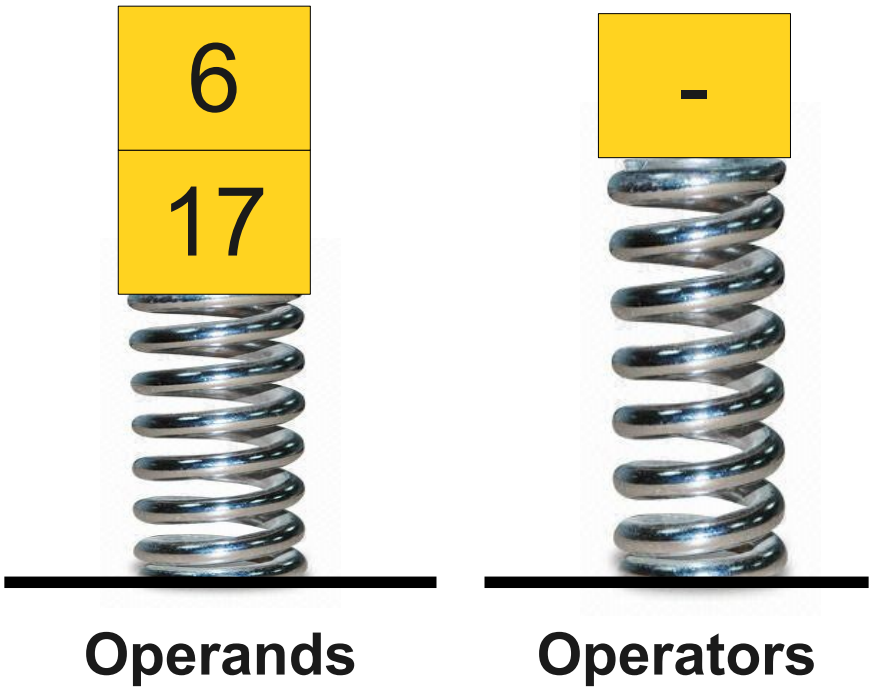
/	2



The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

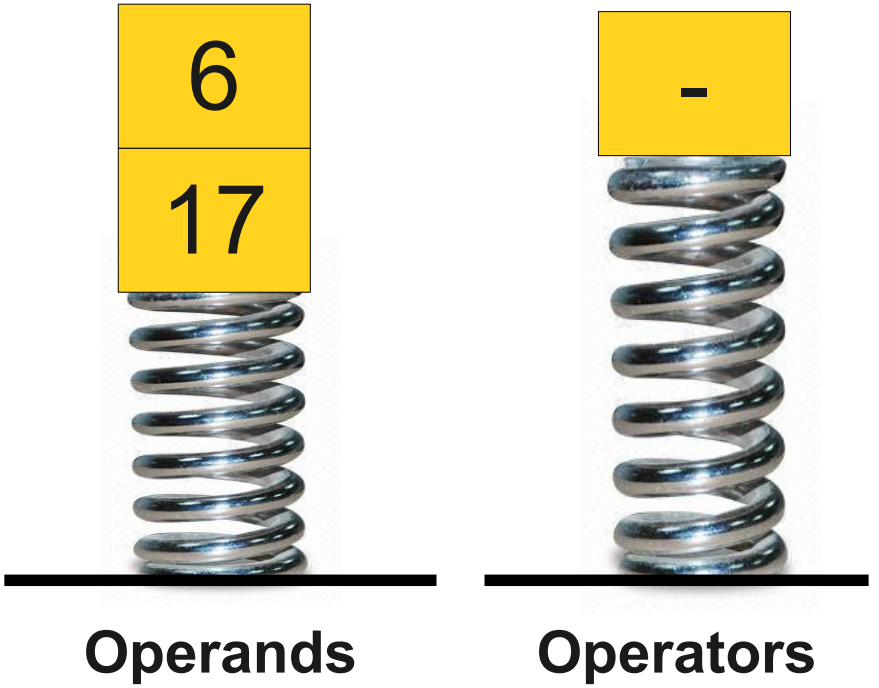
/	2



The Shunting-Yard Algorithm

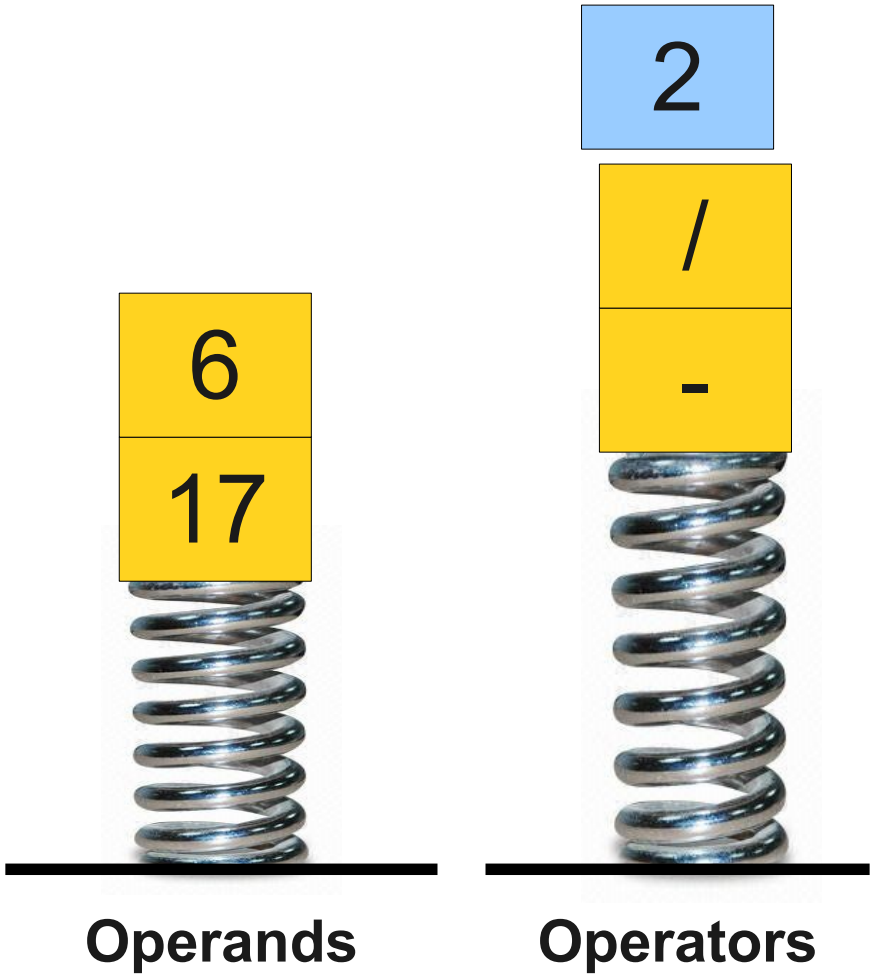
2	+	3	*	5	-	6	/	2

/	2



The Shunting-Yard Algorithm

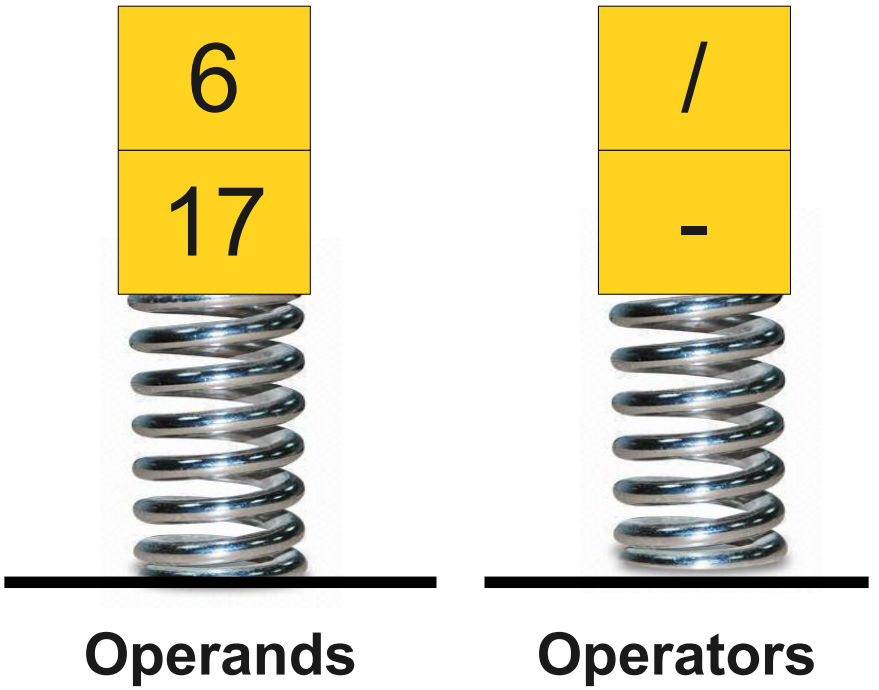
2	+	3	*	5	-	6	/	2



The Shunting-Yard Algorithm

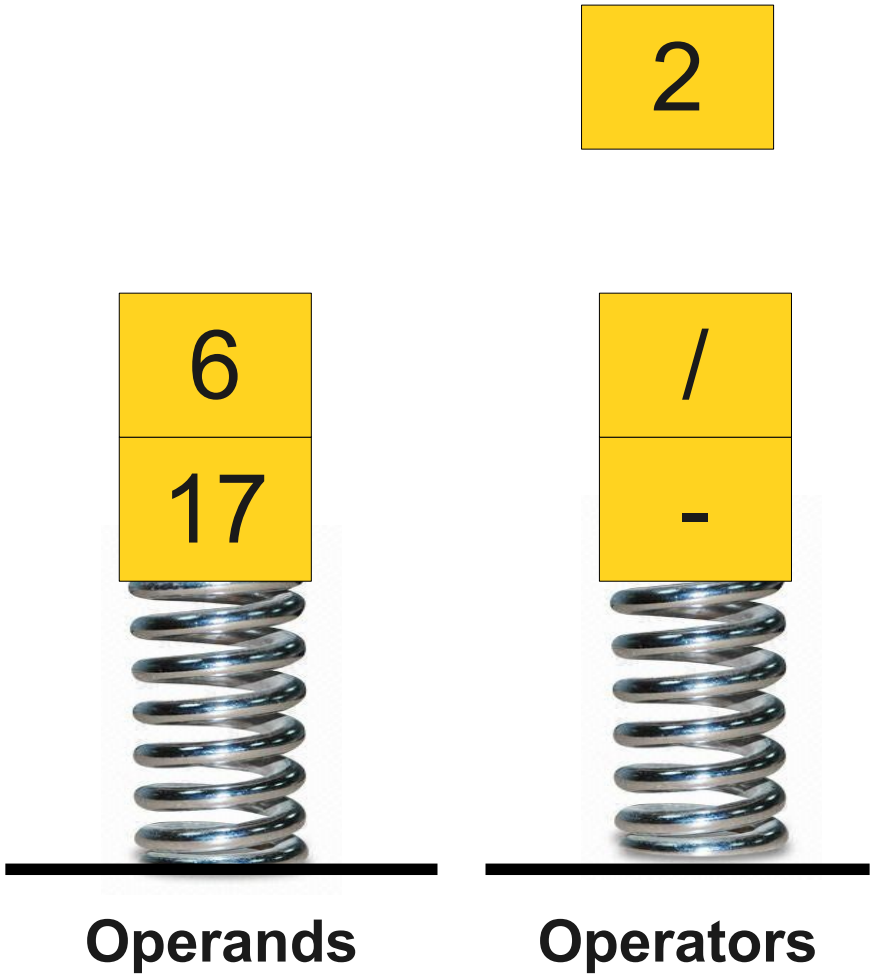
2	+	3	*	5	-	6	/	2

2



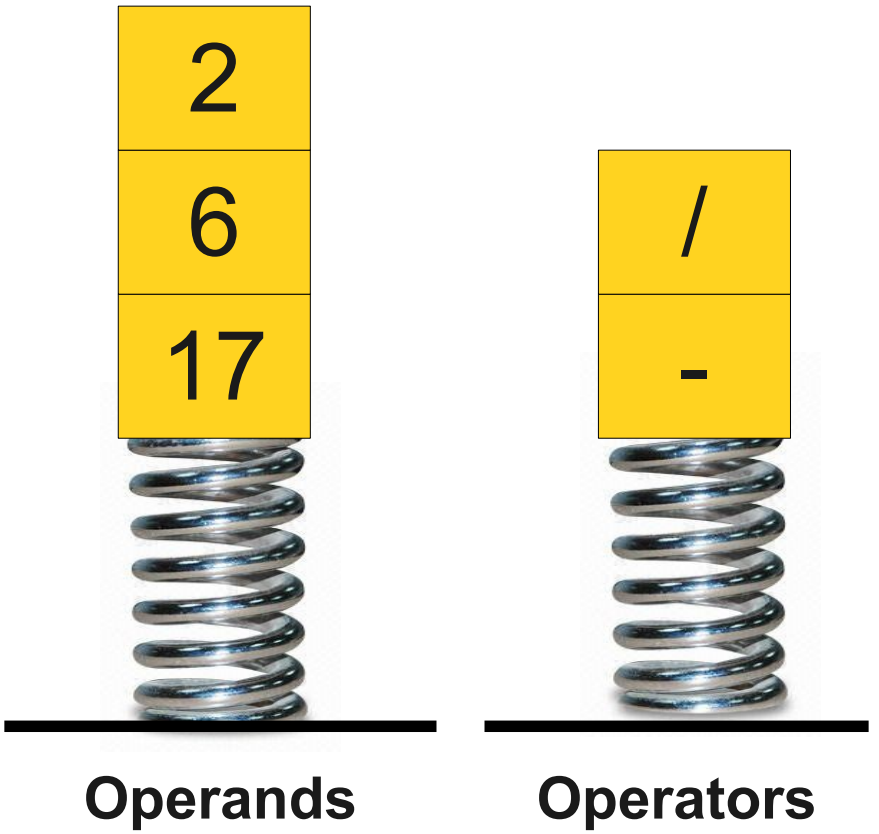
The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2



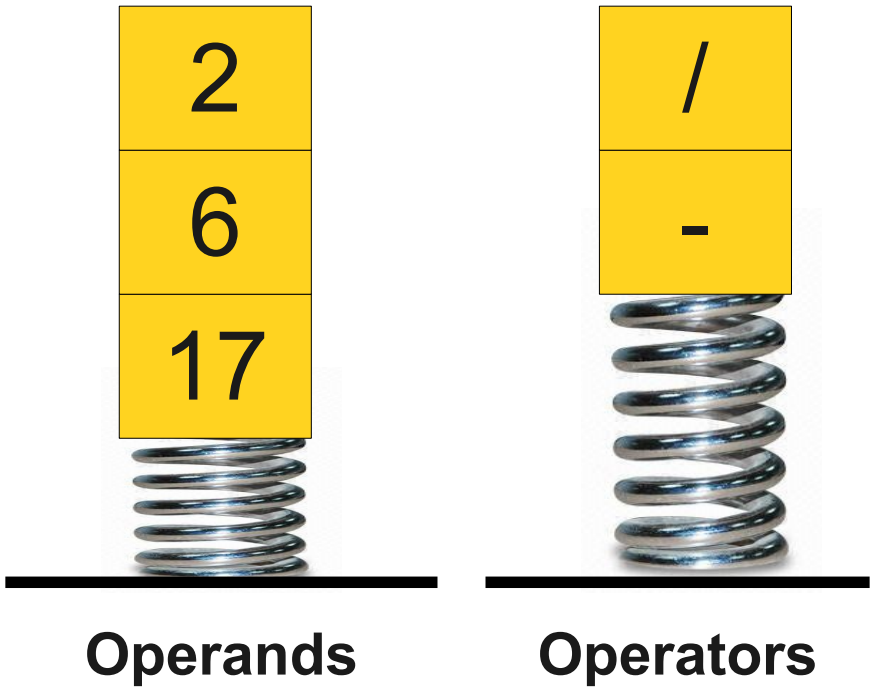
The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2



The Shunting-Yard Algorithm

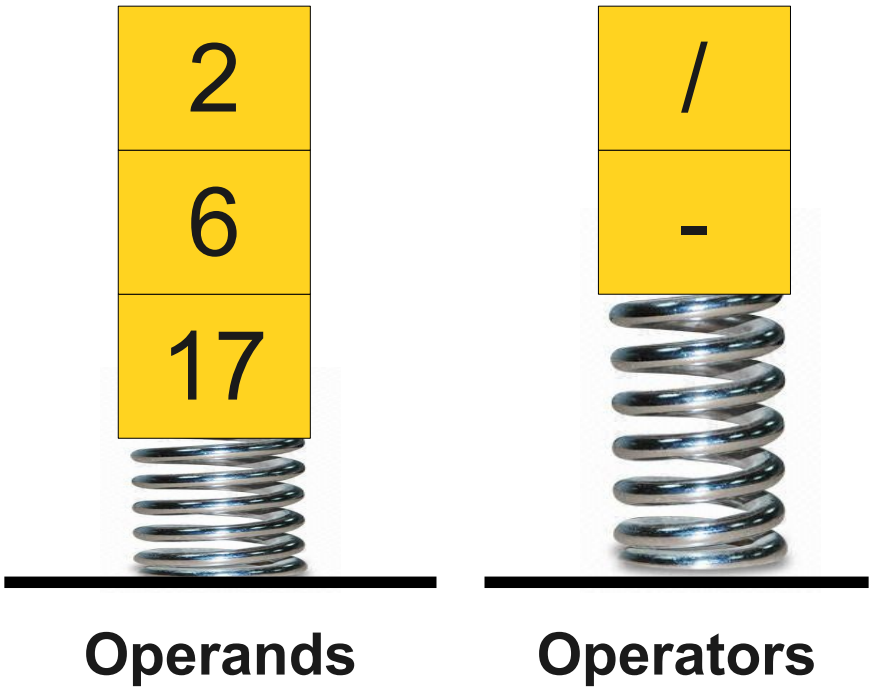
2	+	3	*	5	-	6	/	2



The Shunting-Yard Algorithm

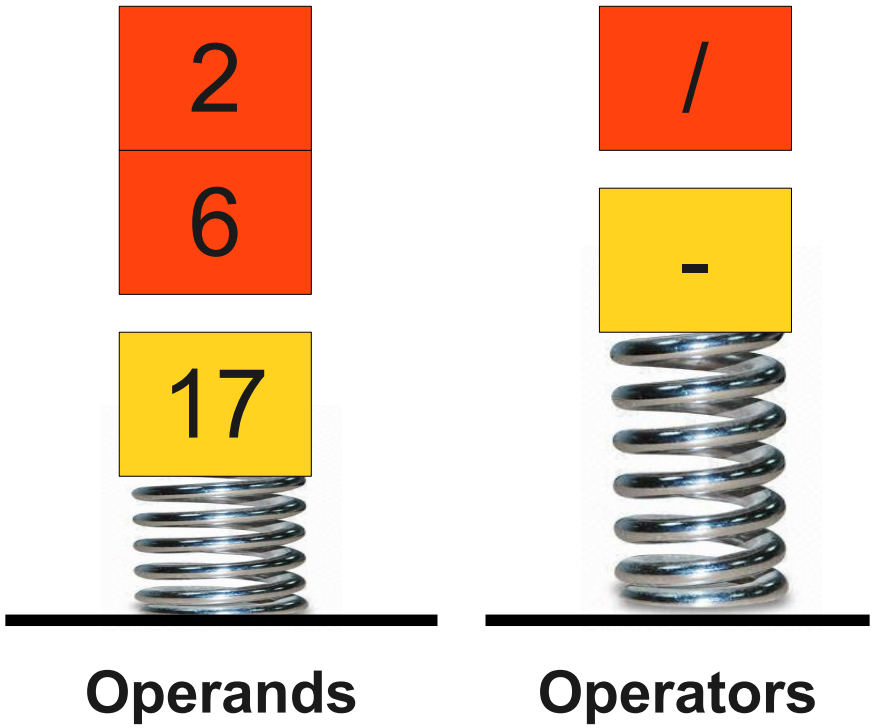
2	+	3	*	5	-	6	/	2

Now that we've read all the tokens, we can finish evaluating all the expressions.



The Shunting-Yard Algorithm

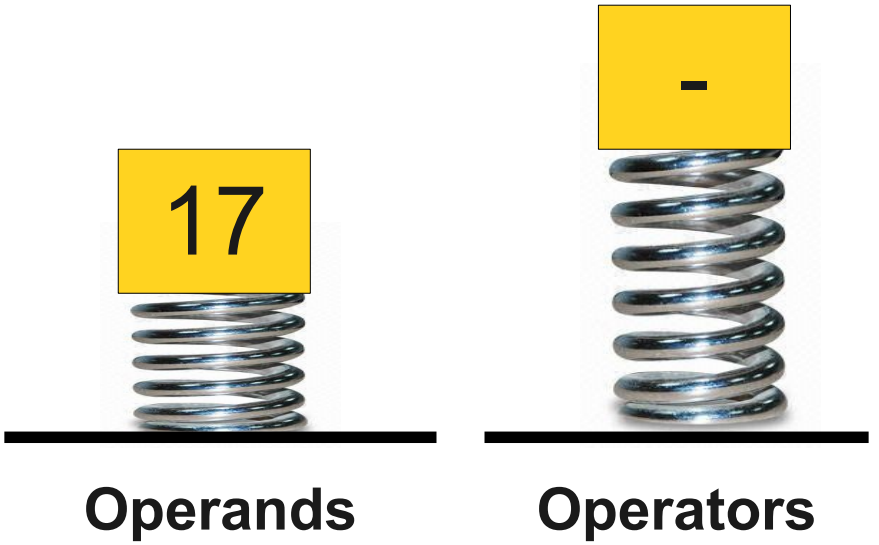
2	+	3	*	5	-	6	/	2



The Shunting-Yard Algorithm

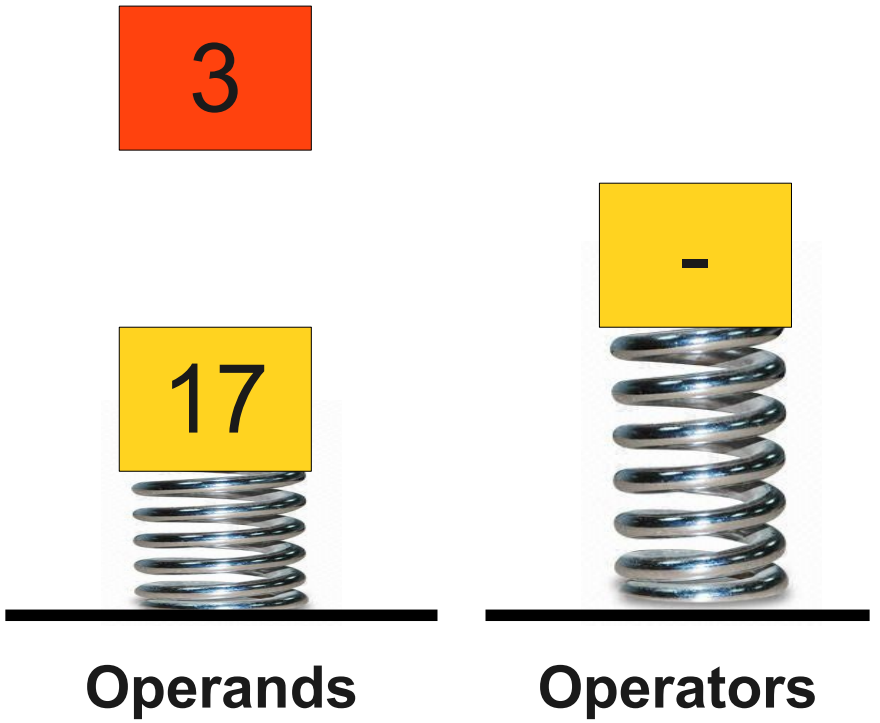
2	+	3	*	5	-	6	/	2

6	/	2



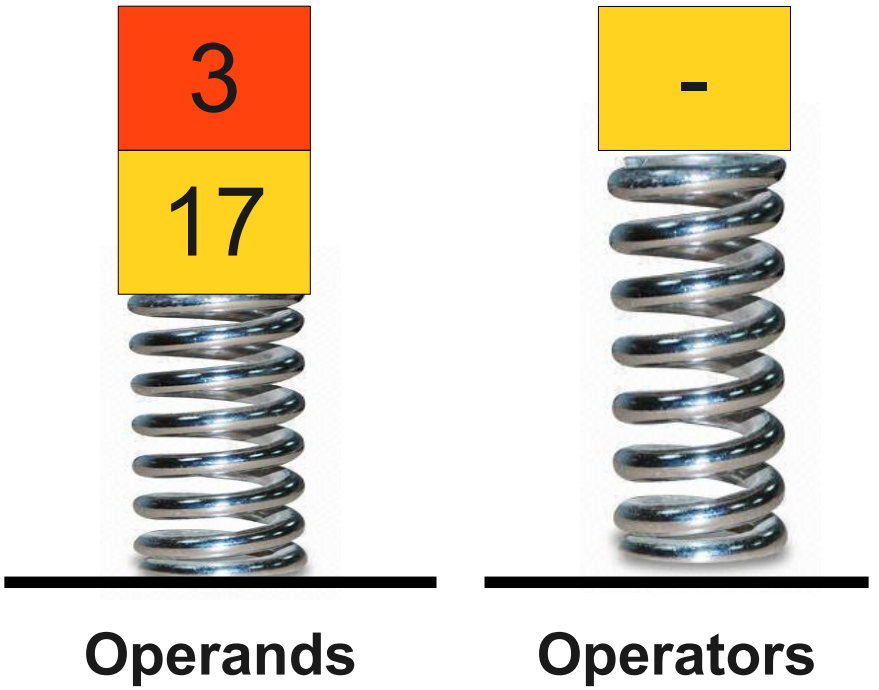
The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2



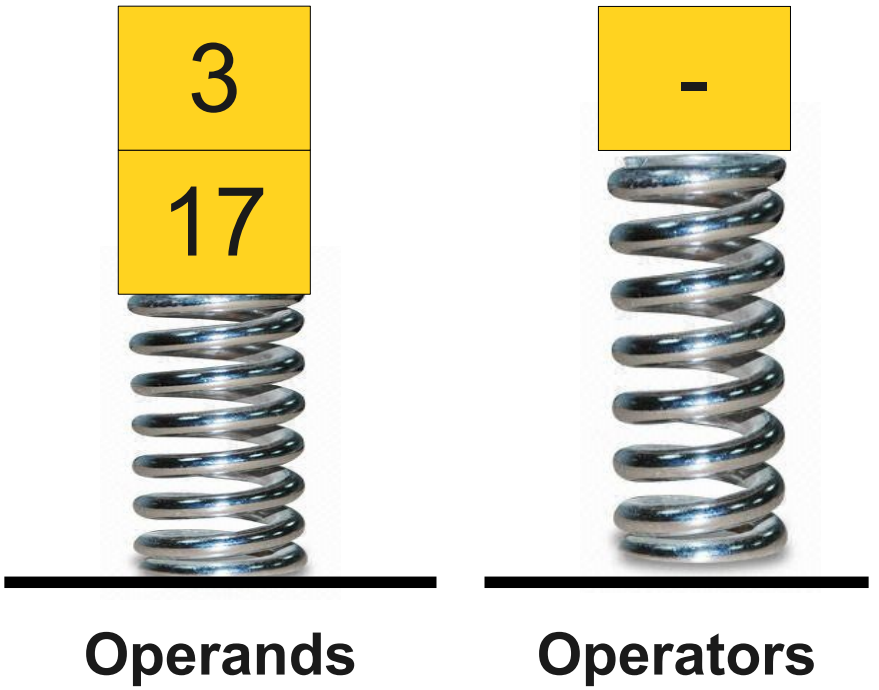
The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2



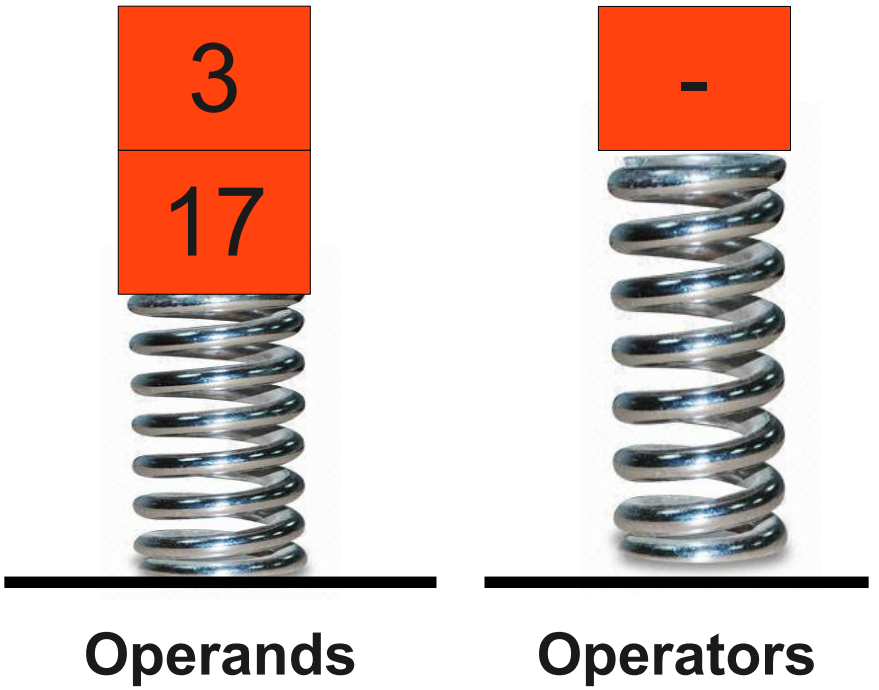
The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2



The Shunting-Yard Algorithm

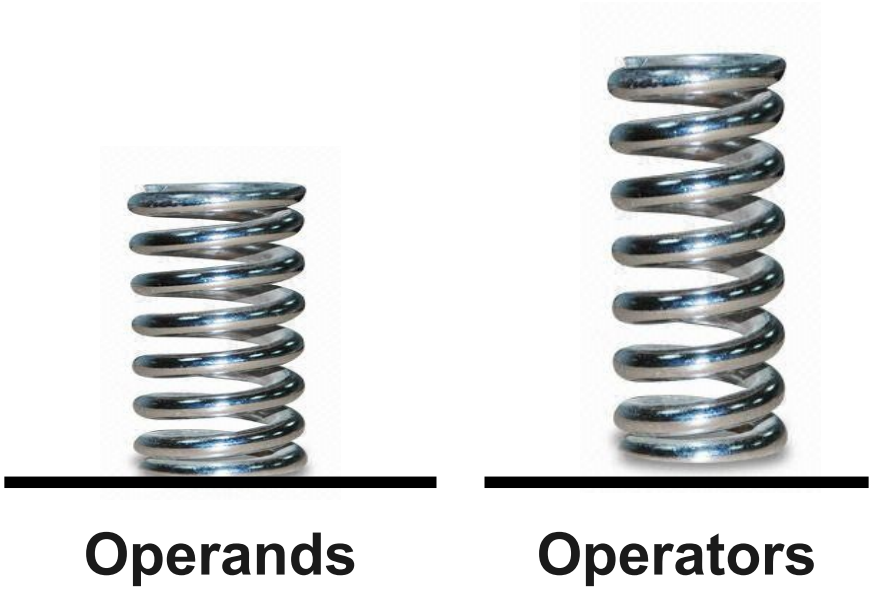
2	+	3	*	5	-	6	/	2



The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2

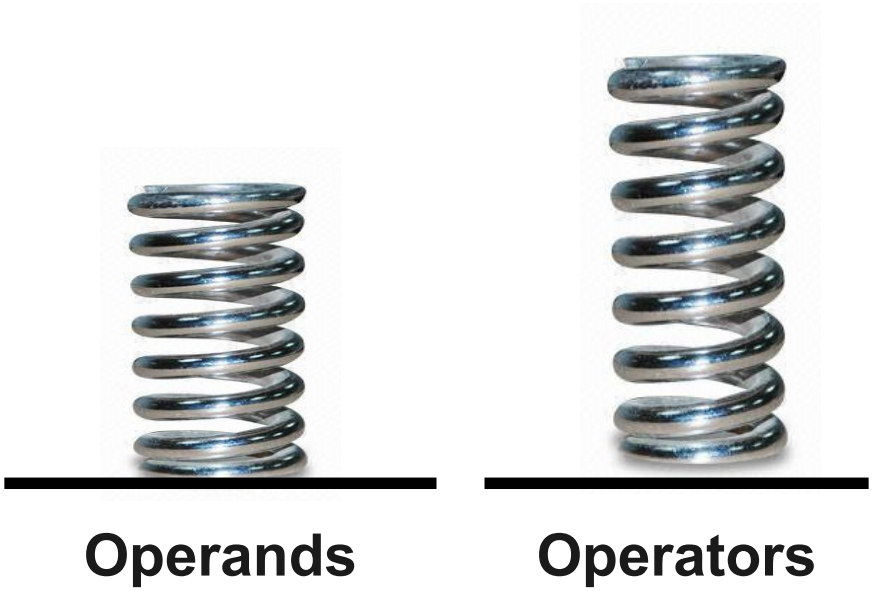
17	-	3



The Shunting-Yard Algorithm

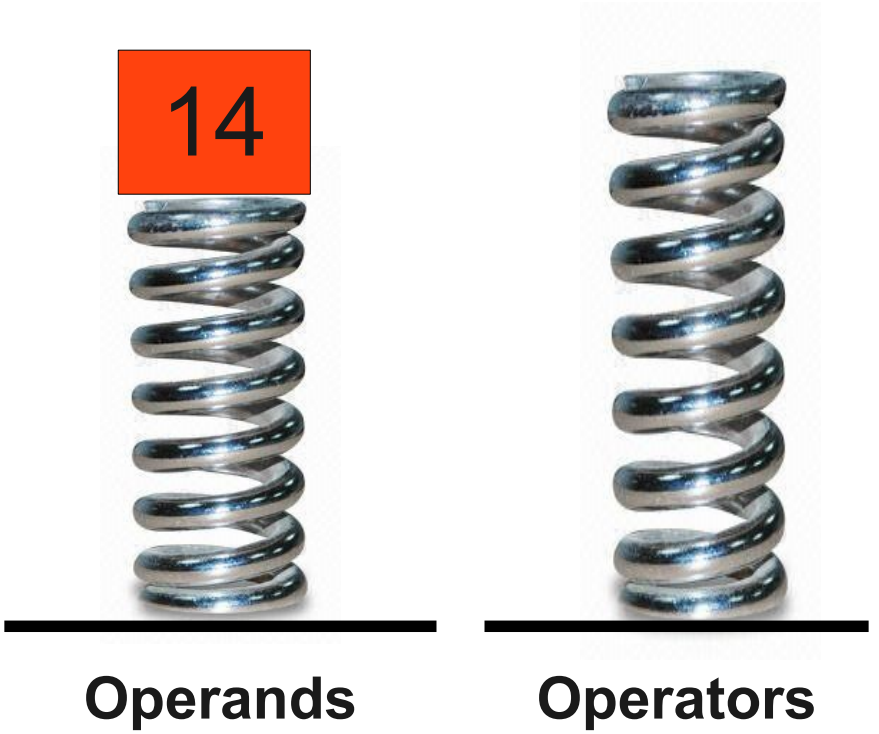
2	+	3	*	5	-	6	/	2

14



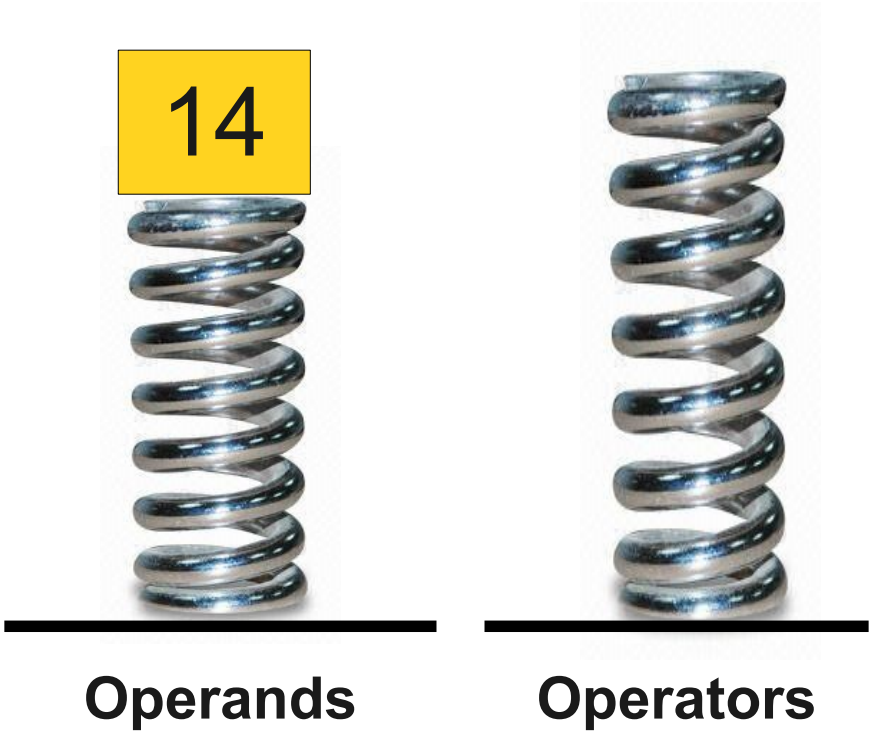
The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2



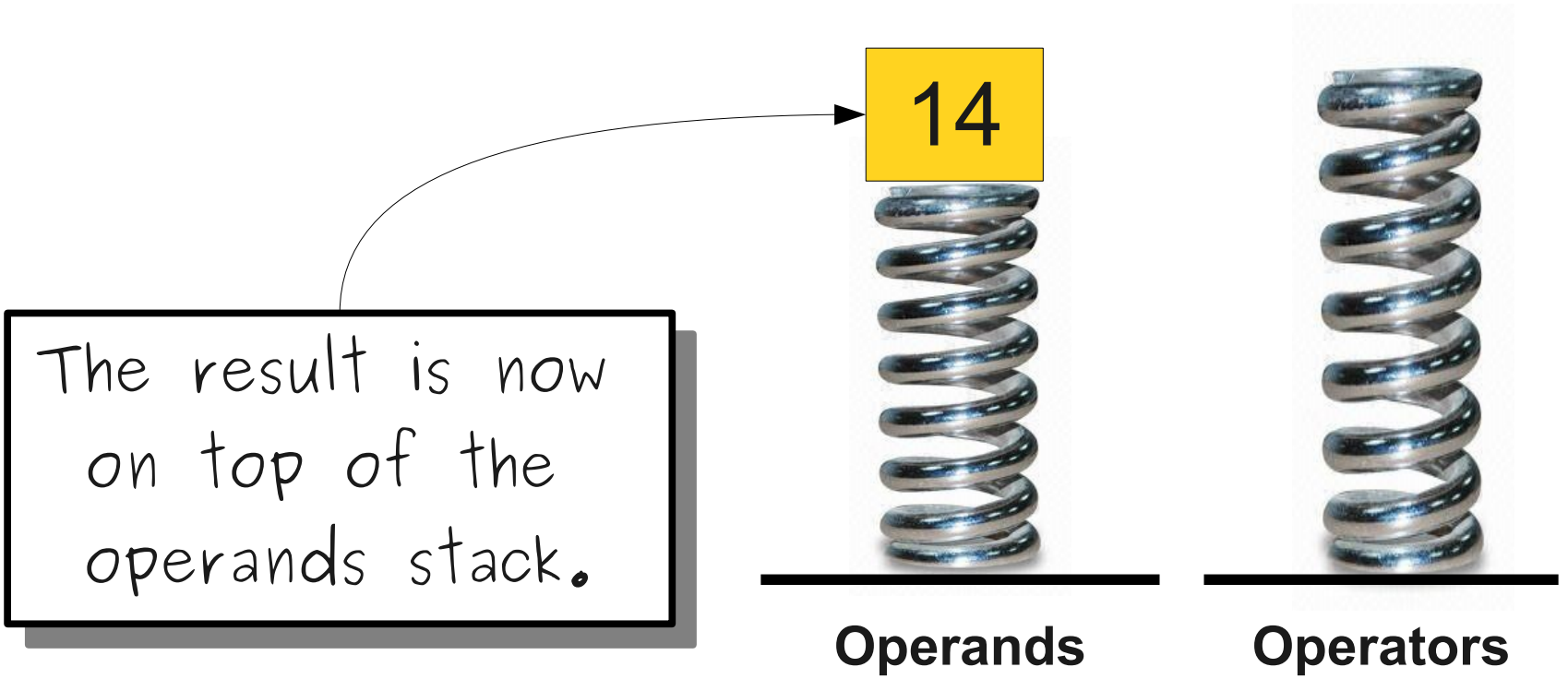
The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2



The Shunting-Yard Algorithm

2	+	3	*	5	-	6	/	2



The Shunting-Yard Algorithm

- Maintain a stack of operators and a stack of operands.
- For each token:
  - If it's a number, push it onto the operand stack.
  - If it's an operator:
    - Keep evaluating operands until the current operator has higher precedence than the most recent operator.
    - Push the operator onto the operator stack.
- Once all input is done, keep evaluating operators until no operators remain.
- The value on the operand stack is the overall result.

TokenScanner

- The **TokenScanner** class can be used to break apart a string into smaller pieces.
- Construct a TokenScanner to piece apart a string as follows:

```
TokenScanner scanner(str);
```

- Configure options (ignore comments, ignore spaces, add operators, etc.)
- Use the following loop to read tokens one at a time:

```
while (scanner.hasMoreTokens()) {
    string token = scanner.nextToken();
    /* … process token … */
}
```

- Check the documentation for more details; there are some really cool tricks you can do with the TokenScanner!

Extensions to Shunting-Yard

- How might you update the shunting-yard algorithm to:
  - Handle/report syntax errors in the input?
  - Support parentheses?
  - Support functions like sin, cos, and tan?
  - Support variables?
- For more information on scanning and parsing, take **CS124** (*From Languages to Information*) or **CS143** (*Compilers*).

Next Time

- **`Vector`**
  - A standard collection for sequences.
- **`Grid`**
  - A standard collection for 2D data.