REVERSE POLISH NOTATION CALCULATOR

Most people have learned to write arithmetic expressions with the operators in-between the numbers (`(2 + 2) / 5`). This is how most calculators let you insert mathematical expressions and probably the notation you were taught to count with in school. This notation has the downside of needing you to know about operator precedence: multiplication and division are more important (have a higher *precedence*) than addition and subtraction.

Another notation exists, called *prefix notation* or *Polish notation*, where the operator comes before the operands. Under this notation, `(2 + 2) / 5` would become `(/ (+ 2 2) 5)`. If we decide to say `+` and `/` always take two arguments, then `(/ (+ 2 2) 5)` can simply be written as `/ + 2 2 5`.

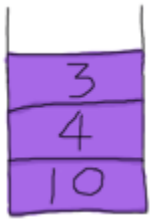
However, we will instead focus on *Reverse Polish notation* (or just *RPN*), which is the opposite of prefix notation: the operator follows the operands. The same example as above in RPN would be written `2 2 + 5 /`. Other example expressions could be `9 * 5 + 7` or `10 * 2 * (3 + 4) / 2` which get translated to `9 5 * 7 +` and `10 2 * 3 4 + * 2 /`, respectively. This notation was used a whole lot in early models of calculators as it would take little memory to use. In fact some people still carry RPN calculators around. We'll write one of these.

First of all, it might be good to understand how to read RPN expressions. One way to do it is to find the operators one by one and then regroup them with their operands by arity:

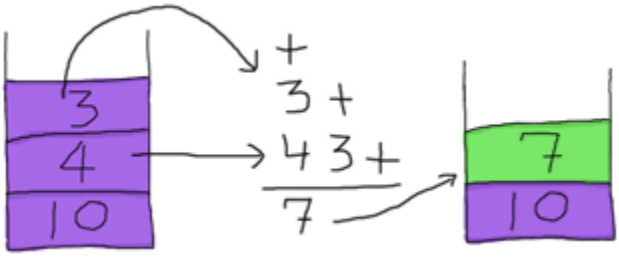
```
10 4 3 + 2 * -

10 (4 3 +) 2 * -

10 ((4 3 +) 2 *) -

(10 ((4 3 +) 2 *) -)

(10 (7 2 *) -)

(10 14 -)

-4
```

However, in the context of a computer or a calculator, a simpler way to do it is to make a *stack* of all the operands as we see them. Taking the mathematical expression `10 4 3 + 2 * -`, the first operand we see is `10`. We add that to the stack. Then there's `4`, so we also push that on top of the

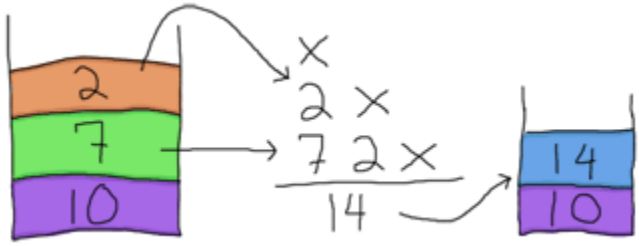
stack. In third place, we have 3; let's push that one on the stack too. Our stack should now look like this:



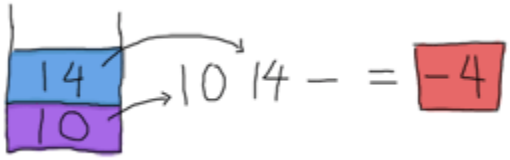
The next character to parse is a `+`. That one is a function of arity 2. In order to use it we will need to feed it two operands, which will be taken from the stack:



So we take that 7 and push it back on top of the stack (yuck, we don't want to keep these filthy numbers floating around!) The stack is now `[7,10]` and what's left of the expression is `2 * -`. We can take the 2 and push it on top of the stack. We then see `*`, which needs two operands to work. Again, we take them from the stack:



And push 14 back on top of our stack. All that remains is `-`, which also needs two operands. O Glorious luck! There are two operands left in our stack. Use them!



And so we have our result. This stack-based approach is relatively fool-proof and the low amount of parsing needed to be done before starting to calculate results explains why it was a good idea for old calculators to use this. There are other reasons to use RPN, but this is a bit out of the scope of this guide, so you might want to hit theWikipedia article instead.

Writing this solution in Erlang is not too hard once we've done the complex stuff. It turns out the tough part is figuring out what steps need to be done in order to get our end result and we just did that. Neat. Open a file named `calc.erl`.

The first part to worry about is how we're going to represent a mathematical expression. To make things simple, we'll probably input them as a string: `"10 4 3 + 2 * -"`. This string has whitespace, which isn't part of our problem-solving process, but is necessary in order to use a simple tokenizer. What would be usable then is a list of terms of the form `["10","4","3","+","2","*","-"]` after going through the tokenizer. Turns out the function `string:tokens/2` does just that:

```
1> string:tokens("10 4 3 + 2 * -", " ").
["10","4","3","+","2","*","-"]
```

That will be a good representation for our expression. The next part to define is the stack. How are we going to do that? You might have noticed that Erlang's lists act a lot like a stack. Using the cons (`|`) operator in `[Head|Tail]` effectively behaves the same as pushing *`Head`* on top of a stack (*`Tail`*, in this case). Using a list for a stack will be good enough.

To read the expression, we just have to do the same as we did when solving the problem by hand. Read each value from the expression, if it's a number, put it on the stack. If it's a function, pop all the values it needs from the stack, then push the result back in. To generalize, all we need to do is go over the whole expression as a loop only once and accumulate the results. Sounds like the perfect job for a fold!

What we need to plan for is the function that `lists:foldl/3` will apply on every operator and operand of the expression. This function, because it will be run in a fold, will need to take two arguments: the first one will be the element of the expression to work with and the second one will be the stack.

We can start writing our code in the `calc.erl` file. We'll write the function responsible for all the looping and also the removal of spaces in the expression:

```
-module(calc).
-export([rpn/1]).

rpn(L) when is_list(L) ->
[Res] = lists:foldl(fun rpn/2, [], string:tokens(L, " ")),
Res.
```

We'll implement `rpn/2` next. Note that because each operator and operand from the expression ends up being put on top of the stack, the solved expression's result will be on that stack. We need to get that last value out of there before returning it to the user. This is why we pattern match over `[Res]` and only return *`Res`*.

Alright, now to the harder part. Our `rpn/2` function will need to handle the stack for all values passed to it. The head of the function will probably look like `rpn(Op,Stack)` and its return value like `[NewVal|Stack]`. When we get regular numbers, the operation will be:

```
rpn(X, Stack) -> [read(X)|Stack].
```

Here, `read/1` is a function that converts a string to an integer or floating point value. Sadly, there is no built-in function to do this in Erlang (only one or the other). We'll add it ourselves:

```
read(N) ->
case string:to_float(N) of
{error,no_float} -> list_to_integer(N);
{F,_} -> F
end.
```

Where `string:to_float/1` does the conversion from a string such as `"13.37"` to its numeric equivalent. However, if there is no way to read a floating point value, it returns `{error,no_float}`. When that happens, we need to call `list_to_integer/1` instead. Now back to `rpn/2`. The numbers we encounter all get added to the stack. However, because our pattern matches on anything (see Pattern Matching), operators will also get pushed on the stack. To avoid this, we'll put them all in preceding clauses. The first one we'll try this with is the addition:

```
rpn("+", [N1,N2|S]) -> [N2+N1|S];
rpn(X, Stack) -> [read(X)|Stack].
```

We can see that whenever we encounter the `"+"` string, we take two numbers from the top of the stack (*N1*,*N2*) and add them before pushing the result back onto that stack. This is exactly the same logic we applied when solving the problem by hand. Trying the program we can see that it works:

```
1> c(calc).
{ok,calc}
2> calc:rpn("3 5 +").
8
3> calc:rpn("7 3 + 5 +").
15
```

The rest is trivial, as you just need to add all the other operators:

```
rpn("+", [N1,N2|S]) -> [N2+N1|S];
rpn("-", [N1,N2|S]) -> [N2-N1|S];
rpn("*", [N1,N2|S]) -> [N2*N1|S];
rpn("/", [N1,N2|S]) -> [N2/N1|S];
rpn("^", [N1,N2|S]) -> [math:pow(N2,N1)|S];
rpn("ln", [N|S])    -> [math:log(N)|S];
rpn("log10", [N|S]) -> [math:log10(N)|S];
rpn(X, Stack) -> [read(X)|Stack].
```

Note that functions that take only one argument such as logarithms only need to pop one element from the stack. It is left as an exercise to the reader to add functions such as 'sum' or 'prod' which return the sum of all the elements read so far or the products of them all. To help you out, they are implemented in my version of `calc.erl` already.

To make sure this all works fine, we'll write very simple unit tests. Erlang's `=` operator can act as an *assertion* function. Assertions should crash whenever they encounter unexpected values, which is exactly what we need. Of course, there are more advanced testing frameworks for Erlang, including Common Test and EUnit. We'll check them out later, but for now the basic `=` will do the job:

```
rpn_test() ->
5 = rpn("2 3 +"),
87 = rpn("90 3 -"),
-4 = rpn("10 4 3 + 2 * -"),
-2.0 = rpn("10 4 3 + 2 * - 2 /"),
ok = try
rpn("90 34 12 33 55 66 + * - +")
catch
error:{badmatch,[_|_]} -> ok
end,
4037 = rpn("90 34 12 33 55 66 + * - + -"),
8.0 = rpn("2 3 ^"),
true = math:sqrt(2) == rpn("2 0.5 ^"),
true = math:log(2.7) == rpn("2.7 ln"),
true = math:log10(2.7) == rpn("2.7 log10"),
50 = rpn("10 10 10 20 sum"),
10.0 = rpn("10 10 10 20 sum 5 /"),
1000.0 = rpn("10 10 20 0.5 prod"),
ok.
```

The test function tries all operations; if there's no exception raised, the tests are considered successful. The first four tests check that the basic arithmetic functions work right. The fifth test specifies behaviour I have not explained yet. The `try ... catch` expects a badmatch error to be thrown because the expression can't work:

```
90 34 12 33 55 66 + * - +

90 (34 (12 (33 (55 66 +) *) -) +)
```

At the end of `rpn/1`, the values `-3947` and `90` are left on the stack because there is no operator to work on the `90` that hangs there. Two ways to handle this problem are possible: either ignore it and

only take the value on top of the stack (which would be the last result calculated) or crash because the arithmetic is wrong. Given Erlang's policy is to let it crash, it's what was chosen here. The part that actually crashes is the `[Res]` in `rpn/1`. That one makes sure only one element, the result, is left in the stack.

The few tests that are of the form `true = FunctionCall1 == FunctionCall2` are there because you can't have a function call on the left hand side of `=`. It still works like an assert because we compare the comparison's result to`true`.

I've also added the test cases for the sum and prod operators so you can exercise yourselves implementing them. If all tests are successful, you should see the following:

```
1> c(calc).
{ok,calc}
2> calc:rpn_test().
ok
3> calc:rpn("1 2 ^ 2 2 ^ 3 2 ^ 4 2 ^ sum 2 -").
28.0
```

Where 28 is indeed equal to `sum(1² + 2² + 3² + 4²) - 2`. Try as many of them as you wish.

One thing that could be done to make our calculator better would be to make sure it raises `badarith` errors when it crashes because of unknown operators or values left on the stack, rather than our current `badmatch` error. It would certainly make debugging easier for the user of the calc module.

Source : http://learnyousomeerlang.com/functionally-solving-problems