Tutorial 3: RPN Calculator

In this tutorial we will build a reverse Polish notation (RPN) calculator. RPN is a way of writing arithmetic expressions that puts the operator **after** its arguments. For example:

regular notation (infix)	reverse Polish notation (postfix)
$1 + 2$	$1\ 2\ +$
$(2 \times 3) + 5$	$2\ 3\ \times\ 5\ +$
$3 \times (5 + 7)$	$3\ 5\ 7\ +\ \times$
$(1 + 2) \times (3 + 4)$	$1\ 2\ +\ 3\ 4\ +\ \times$

Note how in this notation there is no need for parentheses or precedence rules. RPN is easy to represent in a computer, as a list of numbers and arithmetic operators, and easy to evaluate, which we will do in this tutorial.

To evaluate an RPN expression we use a second list, the **stack**, to hold previously evaluated arguments. The stack thus only holds numbers. The expression is then evaluated as follows:

- **Numbers:** a number is pushed to the stack.
- **Operators:** an operator takes its arguments from the stack, applies the operation, and returns the result to the stack.

For example, the last expression above evaluates as below, where each line shows a step in the computation. To avoid shuffling numbers around too much, we depict the stack with the head to the right. The first step pushes 1 to the stack, the second step pushes 2, the third step processes the first $+$ by popping 1 and 2 and pushing back 3, and so on.

stack	expression
	$1\ 2\ +\ 3\ 4\ +\ \times$
1	$2\ +\ 3\ 4\ +\ \times$
$1\ 2$	$+\ 3\ 4\ +\ \times$
3	$3\ 4\ +\ \times$
$3\ 3$	$4\ +\ \times$
$3\ 3\ 4$	$+\ \times$
$3\ 7$	$\times$
21	

We will limit ourselves to three mathematical operations: addition $(+)$, multiplication $(\times)$, and unary negation $(-)$, as in (-3), not subtraction $(5 - 3)$. From here on we will use $(+)$, $(*)$, and $(-)$ as **symbols** for arithmetic operations in RPN expressions, to distinguish them from the actual mathematical operations $(+)$, $(\times)$, and $(-)$.

Formal definition

We will define our RPN calculator **formally**, in mathematics. If you don't find this helpful, you can skip this part, and simply work from the informal description and the example above.

- A **stack** S is a sequence of integers $i_1 \ldots i_n$, defined inductively by

$$S ::= \varepsilon \mid S\ i$$

  where ε is the empty stack, and $S\ i$ adds an integer i to the head of the stack S.

- An **expression** E is a sequence of natural numbers and the symbols `+`, `*`, and `-` for arithmetic operations, defined inductively by

$$E ::= \varepsilon \mid n\ E \mid \texttt{+}\ E \mid \texttt{*}\ E \mid \texttt{-}\ E$$

  where $n \in \mathbb{N}$ is a natural number.

- A **state** is a pair (S, E) of a stack S and an expression E. An **initial** state is one (ε, E) where the stack is empty.

- The **transitions** from one state to another are given by the following top–to–bottom rules:

$$\frac{(\ S\ ,\ n\ E\ )}{(\ S\ n\ ,\ \ E\ )} \qquad \frac{(\ S\ i\ j\ \ ,\ \texttt{+}\ E\ )}{(\ S\ (i{+}j)\ ,\ \ E\ )} \qquad \frac{(\ S\ i\ j\ \ ,\ \texttt{*}\ E\ )}{(\ S\ (i{\times}j)\ ,\ \ E\ )} \qquad \frac{(\ S\ i\ \ ,\ \texttt{-}\ E\ )}{(\ S\ (-i)\ ,\ \ E\ )}$$

- A state is **final** if there are no transitions from it. A **success state** is a final state of the form $(\varepsilon\ i\ ,\ \varepsilon)$, i.e. where the stack holds a single integer i and the expression is empty. A final state that is not a success state is a **failure state**.

- A **run** is a sequence of transitions from an initial to a final state. A run is **successful** if it ends in a success state. A run **for an expression** E is one from the initial state (ε, E). The **return value** of a successful run to a state $(\varepsilon\ i, \varepsilon)$ is the integer i.

We may depict a run as a transition with a double line, for example:

$$\frac{(\ \varepsilon\ \ ,\ E\ )}{\overline{\overline{(\ \varepsilon\ i\ ,\ \ \varepsilon\ )}}}$$

A successful run for an expression E computes the value of E as the return value i.

Implementation

The main aim of our implementation will be a function

```
run :: String -> Int
```

that takes a string representing an RPN expression and evaluates it using a stack. The task may be broken down into the following steps.

- **Tokenizing:** to split the input string into distinct units called **tokens**, each a string representing a number or symbol.
- **Parsing:** to read the tokenized input and format it in abstract syntax (a data type) representing an RPN expression.
- **Evaluating:** to initialize the RPN expression with an empty stack, run the stack calculator until the expression is empty, and return the solution from the stack.

An input string may be expected to represent an RPN expression consisting natural numbers and the operator symbols `+`, `*`, and `-`, separated with spaces. For input strings not representing an RPN expression in this way the function `run` may throw an exception.

```
*Main> run "1 2 +"
3
*Main> run "193 83 *"
16019
*Main> run "1 2 + 3 4 + *"
21
*Main> run "3 -"
-3
*Main> run "100 3 - +"
97
```

Exercise 1: Implement the function `run`. First, try to do everything yourself: break the problem into logical components, and design the necessary types and data types. If you get stuck, follow the guidance on the next page.

Functions you might find useful:

```
read      :: Read a => String -> a
takeWhile :: (a -> Bool) -> [a] -> [a]
dropWhile :: (a -> Bool) -> [a] -> [a]
words     :: String -> [String]
```

Guidance

Please try to approach the exercise by yourself first. If you do get stuck, you can use this guidance for help.

First, it will be helpful to split your problem into the three parts of tokenizing, parsing, and evaluation, and so into three functions `tokenize`, `parse`, and `evaluate`. The function `run` will then combine all three.

The function `tokenize :: String -> [String]` should split the input string into smaller strings, splitting along spaces.

```
*Main> tokenize "34 19  +   0      ++"
["34","19","+","0","++"]
```

Next, we need a representation of expressions as lists of numbers and operator symbols. We can do this by defining a new type `Action` for stack actions, so that RPN expressions have type `[Action]`. You can define `Action` in one of several ways, using new `data` types and/or `type` aliasing with `Either`. To help test your functions, make `Action` an instance of `Show`. The types of the other two functions are then:

```
parse    :: [String] -> [Action]
evaluate :: [Action] -> Int
```

The function `parse` should map each string to the corresponding action.

To implement `evaluate` there are two approaches. **One:** directly as a tail-recursive function with the stack as the accumulator (use an auxiliary function to implement the recursion). **Two:** create a data type `State` as a pair of the stack (a list of integers) and the expression (a list of actions), and implement the following functions:

```
initialize :: [Action] -> State
step       :: State -> State
loop       :: State -> Int
```

The function `step` should take a single step (processing a single `Action`), and `loop` should keep taking steps until the expression is empty, and then return the integer on the stack. At this point it is fine to use exceptions when there are not enough (or too many) items on the stack. Then `evaluate` becomes a combination of `initialize` and `loop`.

Extra

Non-arithmetic operations: RPN expressions are very basic computer programs, consisting of primitive instructions working on the stack. We can use this to add instructions that are outside of normal arithmetic, but are perfectly sensible for operating the stack. We will add these as single-letter instructions. Please add the following:

- `d` to **duplicate** the top item on the stack
- `k` to **kill** (delete) the top item on the stack
- `s` to move the **second** item to the top (swap the top 2 items)
- `t` to move the **third** item to the top (leave the first 2 the same order)

```
*Main> run "6 d *"
36
*Main> run "1 10 s - +"
9
```

```
*Main> run "1 2 3 t k +"
5
```

Type checking: we can check if an RPN expression will produce a correct output without running it, by computing the size of the stack at each step in the computation. For instance, we start with a stack of size 0; pushing an integer increases the size by one; applying a `+` requires at least two elements and decreases the size by one; and so on, while the computation should end with exactly one element on the stack. Checking this is equivalent to type-checking in Haskell; in particular, it can be done at compile-time, and guarantees that the computation will succeed.

Adjust your RPN calculator so that failure to parse gives a “syntax error”, and failure to typecheck gives a “type error”.

```
*Main> run "34*"
*** Exception: Syntax error
```

```
*Main> run "34 *"
*** Exception: Type error
```

Challenges

Avoiding exceptions: to handle exceptions yourself is preferable over using Haskell's exception mechanism. Change the type of `run` to `String -> Either String Int`, where the `String` is an error message and the `Int` is a correct output. Then, add an operation `"/"` for **integer division** $\frac{n}{m}$ on the first two stack items, with a runtime error for division by zero, $\frac{n}{0}$.

```
*Main> run "34*"
Left "Syntax error"
*Main> run "34 *"
Left "Type error"
```

```
*Main> run "2 0 /"
Left "Runtime error"
*Main> run "1 2 + 3 4 + *"
Right 21
```

To avoid exceptions using the `read` function, you may want to use the following functions:

```
isDigit   :: Char -> Bool                  -- module: Data.Char
readMaybe :: Read a => String -> Maybe a   -- module: Text.Read
```

These are not part of the default Haskell functions loaded in `ghci`, but are provided by further **libraries**, called **modules** in Haskell. To use one of these functions, import the relevant library with the following line(s), at the top of your file.

```
import Data.Char (isDigit)
import Text.Read (readMaybe)
```

In parentheses are the functions form the module you wish to import. This can be omitted to import all functions, though this increases the chance of name clashes.

Infix notation: Dijkstra's **shunting yard** algorithm is a simple way of transforming regular (infix) notation into RPN. Look this up on the internet and use it to implement the function

```
runInfix :: String -> Int
```

which accepts strings of regular infix notation, without spaces between tokens, again with the symbols `+` and `*` for binary addition and `-` for (prefix) unary negation.

```
*Main> runInfix "(10+20)*(-30+-40)"
-2100
```

If you have done the last **extra** assignment, you could instead use an `Either` type:

```
runInfix :: String -> Either String Int
```