Let's write a parser for expressions in prefix notation. Here are examples:

`*+42 17!5`	$\rightarrow$	$(42 + 17) \times !5$
`+4-3 +10-5`	$\rightarrow$	$(4 + 3 - (10 + (-5)))$

We will need built-in functions to detect a space or a digit character, so let's import these.

```haskell
module Prefix
where
import Data.Char (isSpace, isDigit, digitToInt)
```

Evaluating a prefix expression requires two steps:

- parsing the expression into *tokens*,
- evaluating these tokens according to the rules of the prefix notation.

1 Tokens, and how to evaluate them

Let's define the possible tokens for our prefix notation. A token can be:

- A number,
- An operator representing an unary function (e.g. factorial)
- An operator representing a binary function (e.g. multiplication)

```haskell
type Number = Integer
data Token = Num Number
  | Op1 (Number → Number)
  | Op2 (Number → Number → Number)
```

Since an unary operator should be followed by another expression, and a binary operator by two expressions, it is natural to represent a parsed expressionas as a list of tokens. For example parsing the expression `*+42 17!5` should result in the following list:

```haskell
example = [Op2 (∗), Op2 (+), Num 42, Num 17, Op1 (λn → product [1 .. n]), Num 5]
```

To evaluate a list of tokens representing a prefix expression, we need to examine the token at the head of the list. If this token matches the pattern `Num n`, then the value is n and the rest of the list is to be evaluated further.
If the head of the list matches an unary operator, `Op1 f`, we have to apply the function f to the value represented by the rest of the list.
If the head of the list matches a binary operator, `Op2 f`, then we have to first evaluate the rest of the list, which will give us the first operand value n and a remainging list, and then the evaluation amounts to evaluating a list starting with the (unary) partial application $f n$ to the value given by the rest of the list.
Finally, evaluating an empty list should yield the (arbitrary) value 0, and an empty list.

```haskell
eval :: [Token] → (Number, [Token])
eval [ ] = (0, [ ])
eval (Num n : ts) = (n, ts)
eval (Op1 f : ts) = (f n, ts′) where (n, ts′) = eval ts
eval (Op2 f : ts) = eval (Op1 (f n) : ts′) where (n, ts′) = eval ts
```

Thus the expression `fst (eval example)` should yield 7080.
Here's how the expression `eval [Op2 (+),Num 42, Num 17]` is evaluated:

```
eval [Op2 (+),Num 42, Num 17]
eval (Op1 ((+) n):ts') where (n,ts') = eval [Num 42,Num 17]
eval (Op1 ((+) n):ts') where (n,ts') = (42, [Num 17])
eval (Op1 ((+) 42):[Num 17])
((+) 42 n,ts') where (n,ts') = eval [Num 17]
((+) 42 n,ts') where (n,ts') = (17,[])
((+) 42 17,[])
(59,[])
```

2 Parsing a prefix expression

A parser is a function that scans a string and recognizes a token, or a sequence of tokens, or just anything else.

$$\textbf{newtype}\ \mathit{Parser}\ a = \mathit{Parser}\ \{\mathit{parse} :: \mathit{String} \rightarrow [(a, \mathit{String})]\}$$

We can parse a digit, converting it to its number value:

$$\begin{array}{l}
\mathit{digit} :: \mathit{Parser}\ \mathit{Number} \\
\mathit{digit}\ [\,] = [\,] \\
\mathit{digit}\ (c : cs)\ |\ \mathit{isDigit}\ c = [(\mathit{toNumber}\ c, cs)] \\
\quad \textbf{where} \\
\quad \mathit{toNumber} = \mathit{fromIntegral} \circ \mathit{digitToInt}
\end{array}$$

If we define the class of our parser as a monad, we can then chain effects on the result of a parser:

$$\begin{array}{l}
\textbf{instance}\ \mathit{Monad}\ \mathit{Parser}\ \textbf{where} \\
\quad m \gg\!= k = \mathit{Parser}\ \$\ \lambda s \rightarrow [(y, s'')\ |\ (x, s') \leftarrow \mathit{parse}\ m\ s, (y, s'') \leftarrow \mathit{parse}\ (k\ x, s')]
\end{array}$$