Let's write a parser for expression in prefix notation. We will need built-in functions to detect a space or a digit character.

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

First let's define some operators that our parser will recognize.

```
sPlus = '+'
sMinus = '-'
sMult = '*'
sDiv  = '/'
sMod = '%'
sNegate = '~'
sFact  = '!'
```

A *parser* is a function that breaks a string into components called *tokens*.

```
type Parser = String → [(Token, String)]
```

Typically, a function of type `Parser` will examine its given String argument looking for a specific token. If the token is found, it will be returned in a list, along with the part of the string that remains to be parsed. If the token is not present, an empty list is returned. The token that can be parsed in a prefix expression is one of:

- A number,
- An operator representing an unary function, followed by its operand,
- A operator representing a binary function, followed by its two parameters.

Each operand can be in turn, a full prefix expression. For instance, the expression `*+ 42 17 !5` can be parsed into the `PrefExp` value:

```
Op2 (*)
  (Op2 (+)
     (Num 42)
     (Num 17))
  (Op1 (!)
     (Num 5))
```

Since functions cannot be shown (try to enter `(+)` at the *ghci* prompt to see why), we put additional information in the definition of `PrefExp` values so that showing them make sense.

```
type Number = Integer
type Token = PrefExp
type Symbol = Char
data PrefExp = Num Number
  | Op1 Symbol (Number → Number) PrefExp
  | Op2 Symbol (Number → Number → Number) PrefExp PrefExp
```

To make values of the type `PrefExp` visible (in ghci for example), we define its show function:

```
instance Show PrefExp where
  show (Num n) = show n
  show (Op1 c _ prefExp) = (c : "  ") ++ show prefExp
```

$show\ (Op2\ c\ \_\ prefExp1\ prefExp2) =$
$\quad (c : \texttt{"\ "}) +\!\!+\ show\ prefExp1 +\!\!+ \texttt{"\ "} +\!\!+\ show\ prefExp2$

Our first parser should recognize a digit, convert that value from `Int` and return the `Num` token.

$digit :: Parser$
$digit\ (c : s)\ |\ isDigit\ c = [(digitToNum\ c, s)]$
$\quad \textbf{where}\ digitToNum = Num \circ fromIntegral \circ digitToInt$
$digit\ \_ = [\,]$

Another parser converts all successive digits into a `Num` value.

0	4807
$0 \times 10 + 4$	807
$(0 \times 10 + 4) \times 10 + 8$	07
$((0 \times 10 + 4) \times 10 + 8)$	07
$(((0 \times 10 + 4) \times 10 + 8) \times 10 + 0)$	7
$((((0 \times 10 + 4) \times 10 + 8) \times 10 + 0) \times 10 + 7)$	

$accum :: Integer \to Parser$
$accum\ acc\ s = \textbf{case}\ digit\ s\ \textbf{of}$
$\quad [\,] \to [(Num\ acc, s)]$
$\quad [(Num\ d, s')] \to accum\ (acc * 10 + d)\ s'$

To parse a number, ignore spaces, then if one digit is found, accumulate all the following digits into a number. Otherwise if no digit was found, yield the empty result.

$number :: Parser$
$number\ (c : s)\ |\ isSpace\ c = number\ s$
$number\ (c : s)\ |\ isDigit\ c = accum\ 0\ (c : s)$
$number\ \_ = [\,]$

A parser for negation should recognize the symbol `'~'` and yield an unary operator token with the matching function. The same should be done for the symbol `'!'` and the factorial operation. This can be generalized into a parser for any unary operator.

$unaryOp :: Symbol \to (Number \to Number) \to Parser$
$unaryOp\ op\ f\ (c : s)\ |\ c \equiv op = [(Op1\ op\ f, s)]$
$unaryOp\ \_\ \_\ \_ = [\,]$

$negation :: Parser$
$negation = unaryOp\ \texttt{'\~{}'}\ negate$

$factorial :: Parser$
$factorial = unaryOp\ \texttt{'!'}\ (\lambda n \to product\ [1\,..\,n])$

We can combine two parsers in a way such that one or the other token can be recognized.

$\textbf{infix}\ 2 <|>$
$(<|>) :: Parser \to Parser \to Parser$
$parserA <|> parserB = \lambda s \to \textbf{let}\ result = parserA\ s\ \textbf{in case}\ result\ \textbf{of}$
$\quad [\,] \to parserB\ s$
$\quad \_ \to result$