They associate to the right.

```
a -> a -> a
is just the same as
a -> (a -> a)
```

 \dots be a member of both Num and Ord since Num is not a type of Ord.

Use sections:

Surround the infix function with parens and only supply an

argument on one side but not the other.

- means negative before it means minus.

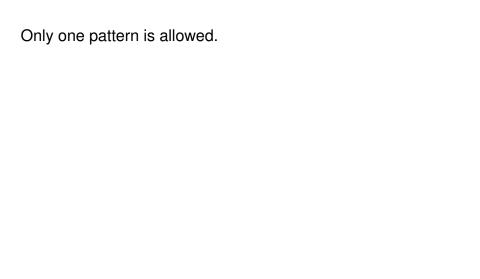
Use subtract instead of -.

See what Haskell infers using :t.

```
ghci> :t flip
flip :: (a -> b -> c) -> b -> a -> c
```

It returns an Int instead of a Num a.

The function body will extend to the end of the line.



They are like fold1/foldr, but they report all intermediate

accumulator states as a list. scan1 places the final result last, scanr puts it first. They do not require explicit starting values. They assume the first (or last, depending on direction) value as the starting

value.

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

It's function application with the lowest (as opposed to the highest) precedence. It is typically used to introduce right association to minimize the number of needed parens.

```
(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c
f . g = \x \rightarrow f (g x)
```

... cannot be determined from the name of a function alone.

The functions must be partially applied until each function just

takes one function.

Using partial application in a function whose result is the argument to another function.

```
fn x = ceiling (negate (tan (cos (max 50 x))))
fn = ceiling . negate . tan . cos . max 50
```

Using a lambda instead of a partially applied function when passing a function to a higher-order method.

If the only purpose of the lambda is to introduce a parameter

If the only purpose of the lambda is to introduce a parameter to a function, you probably could have left it off and used partial application instead.