You can see that polymorphism is Haskell's way of providing generality.

- Parameter is [], x
- Tactum [3, 4, 5] where all parameters (and even return values) can have any of a multitude of types. E.g.,
  - Either because nothing in this function determines the type of the list elements.
  - Type is [a] = [] Tactum x + Tactum []

A function to count the elements in a list:

Polymorphism

In this way, is polymorphic: it can have any of a multitude of types.

- Has a type [a] when alone
- []

If the context does not impose any type on a, it remains uninstantiated.

Recall that a type variable begins in lower case. (An actual type begins in upper case.)

Note that a type variable begins in lower case. (An actual type begins in upper case.)

Polymorphism

How could this be possible?

- The second expression requires [] to have type [Shape].
- The first expression requires [] to have type [Bool].

[Shape] :: [a] = [] True [Bool]

Whatever type [] has, it must be consistent with these:

[Shape] :: []
[Shape] :: [a]
[Bool] :: [a]

Recall the types of [Shape]

- x = x * x
- where
  - opp x = 2
  - x ^ 10 < x
  - x

Bindings in where clauses are visible in guards:

<table>
<thead>
<tr>
<th>x</th>
<th>opp x = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>1 = 10 &lt; x</td>
</tr>
<tr>
<td>V</td>
<td>x = x * x</td>
</tr>
</tbody>
</table>

Here is a slick way to write it, using guards:

| x = x * x then 1 else if x > 0 then -1 else 0 |

A function to find the „sign“ of a number:
function that does \( f(x) \) and returns a slightly modified parameter and even return functions as return values. Such a function is called a higher-order function.

In general, functional languages allow a function to take functions as parameters. And even return functions as return values. Such a function is called a higher-order function.

\[
\text{map} \quad (a \mapsto b) \quad \mapsto \quad [a] \mapsto [b]
\]

The function

**Higher-Order Functions**

This gets boring after a few more examples. Isn't there a better way?

```haskell
-- A data type that represents a value and its square.
data Square = Square Int Int

-- A function that squares a list of integers.
squareList :: [Int] -> [Square]
squareList xs = map Square (map square xs)
```

```haskell
-- A function that squares a list of integers.
squareList :: [Int] -> [Square]
squareList xs = map Square (map square xs)
```

```haskell
-- A function that squares a list of integers.
squareList :: [Int] -> [Square]
squareList xs = map Square (map square xs)
```

```haskell
-- A function that squares a list of integers.
squareList :: [Int] -> [Square]
squareList xs = map Square (map square xs)
```

```haskell
-- A function that squares a list of integers.
squareList :: [Int] -> [Square]
squareList xs = map Square (map square xs)
```

```haskell
-- A function that squares a list of integers.
squareList :: [Int] -> [Square]
squareList xs = map Square (map square xs)
```

```haskell
-- A function that squares a list of integers.
squareList :: [Int] -> [Square]
squareList xs = map Square (map square xs)
```

```haskell
-- A function that squares a list of integers.
squareList :: [Int] -> [Square]
squareList xs = map Square (map square xs)
```

```haskell
-- A function that squares a list of integers.
squareList :: [Int] -> [Square]
squareList xs = map Square (map square xs)
```

```haskell
-- A function that squares a list of integers.
squareList :: [Int] -> [Square]
squareList xs = map Square (map square xs)
```

```haskell
-- A function that squares a list of integers.
squareList :: [Int] -> [Square]
squareList xs = map Square (map square xs)
```

```haskell
-- A function that squares a list of integers.
squareList :: [Int] -> [Square]
squareList xs = map Square (map square xs)
```

```haskell
-- A function that squares a list of integers.
squareList :: [Int] -> [Square]
squareList xs = map Square (map square xs)
```

```haskell
-- A function that squares a list of integers.
squareList :: [Int] -> [Square]
squareList xs = map Square (map square xs)
```

```haskell
-- A function that squares a list of integers.
squareList :: [Int] -> [Square]
squareList xs = map Square (map square xs)
```

```haskell
-- A function that squares a list of integers.
squareList :: [Int] -> [Square]
squareList xs = map Square (map square xs)
```

```haskell
-- A function that squares a list of integers.
squareList :: [Int] -> [Square]
squareList xs = map Square (map square xs)
```

```haskell
-- A function that squares a list of integers.
squareList :: [Int] -> [Square]
squareList xs = map Square (map square xs)
```

```haskell
-- A function that squares a list of integers.
squareList :: [Int] -> [Square]
squareList xs = map Square (map square xs)
```

```haskell
-- A function that squares a list of integers.
squareList :: [Int] -> [Square]
squareList xs = map Square (map square xs)
```

```haskell
-- A function that squares a list of integers.
squareList :: [Int] -> [Square]
squareList xs = map Square (map square xs)
```

```haskell
-- A function that squares a list of integers.
squareList :: [Int] -> [Square]
squareList xs = map Square (map square xs)
```

```haskell
-- A function that squares a list of integers.
squareList :: [Int] -> [Square]
squareList xs = map Square (map square xs)
```

```haskell
-- A function that squares a list of integers.
squareList :: [Int] -> [Square]
squareList xs = map Square (map square xs)
```

```haskell
-- A function that squares a list of integers.
squareList :: [Int] -> [Square]
squareList xs = map Square (map square xs)
```

```haskell
-- A function that squares a list of integers.
squareList :: [Int] -> [Square]
squareList xs = map Square (map square xs)
```

```haskell
-- A function that squares a list of integers.
squareList :: [Int] -> [Square]
squareList xs = map Square (map square xs)
```

```haskell
-- A function that squares a list of integers.
squareList :: [Int] -> [Square]
squareList xs = map Square (map square xs)
```

```haskell
-- A function that squares a list of integers.
squareList :: [Int] -> [Square]
squareList xs = map Square (map square xs)
```

```haskell
-- A function that squares a list of integers.
squareList :: [Int] -> [Square]
squareList xs = map Square (map square xs)
```

```haskell
-- A function that squares a list of integers.
squareList :: [Int] -> [Square]
squareList xs = map Square (map square xs)
```

```haskell
-- A function that squares a list of integers.
squareList :: [Int] -> [Square]
Map
```