Parametric Polymorphism
Parametric Polymorphism

• is a way to make a language **more expressive**, while still maintaining full **static type-safety** (every Haskell expression has a type, and types are all checked at *compile-time*; programs with type errors will not even compile)

• using parametric polymorphism, a function or a data type can be written **generically** so that it can handle values **identically without depending on** their type

• such functions and data types are called **generic functions** and **generic datatypes**
Polymorphism in Haskell

• Two kinds of polymorphism in Haskell – parametric and *ad hoc* (coming later!)

• Both kinds involve **type variables**, standing for arbitrary types.
  • Easy to spot because they start with lower case letters
  • Usually we just use one letter on its own, e.g. a, b, c

• When we use a polymorphic function we will usually do so at a specific type, called an **instance**. The process is called **instantiation**.
Identity function

Consider the identity function:

\[ \text{id } x = x \]

```
Prelude> :t id
id :: a -> a
```

It does not do anything with the input other than returning it, therefore it places no constraints on the input's type.
Prelude> :t id
id :: a -> a

Prelude> id 3
3

Prelude> id "hello"
"hello"

Prelude> id 'c'
'c'
Polymorphic datatypes

• The identity function is the simplest possible polymorphic function but not very interesting

• Most useful polymorphic functions involve **polymorphic types**

• **Notation** – write the name(s) of the type variable(s) used to the left of the = symbol
Maybe

\[
\text{data } \text{Maybe } a = \text{Nothing } \mid \text{Just } a
\]

• \( a \) is the type variable

• When we instantiate \( a \) to something, e.g. \text{String}, the second part of the sum type will be e.g. \text{Just String}

\[
\text{data } \text{Maybe String } = \text{Nothing } \mid \text{Just String}
\]
Pairs

data (,) a b = (,) a b

Note that we usually write (,) a b as (a, b)

More than one type variable

Now let’s write a polymorphic function with it...
Pairs

fst :: (a, b) -> a
fst (x,_) = x

Prelude> fst ("hello","c")
"hello"

Prelude> fst ("hello","world")
"hello"
Lists

data [] a = [] | a : [a]

Note that we usually write [] a as [a], and that we have lots of other syntactic sugar to make list programming more pleasant.

When we are designing a function that manipulates lists, how do we decide whether to be parametric polymorphic?

Q: Does my function depend on the type of the list elements?
<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>:</code></td>
<td>a -&gt; [a] -&gt; [a]</td>
<td>Add a single element to the front: 3:[2,3] → [3,2,3]</td>
</tr>
<tr>
<td><code>++</code></td>
<td>[a] -&gt; [a] -&gt; [a]</td>
<td>Join two lists together: &quot;Ron&quot;++&quot;aldo&quot; → &quot;Ronaldo&quot;</td>
</tr>
<tr>
<td><code>!!</code></td>
<td>[a] -&gt; Int -&gt; a</td>
<td>Return the nth element, counting from 0: [14,3,7]!!1 → 3</td>
</tr>
<tr>
<td><code>concat</code></td>
<td>[[a]] -&gt; [a]</td>
<td>Concatenate a list of lists into a single list: [[2,3],[[],[4]]] → [2,3,4]</td>
</tr>
<tr>
<td><code>length</code></td>
<td>[a] -&gt; Int</td>
<td>The length of a list: length &quot;word&quot; → 4</td>
</tr>
<tr>
<td><code>head</code>, <code>last</code></td>
<td>[a] -&gt; a</td>
<td>The first/last element: head &quot;word&quot; → 'w', last &quot;word&quot; → 'd'</td>
</tr>
<tr>
<td><code>tail</code>, <code>init</code></td>
<td>[a] -&gt; [a]</td>
<td>All but the first/last element: tail &quot;word&quot; → &quot;ord&quot;, init &quot;word&quot; → &quot;wor&quot;</td>
</tr>
<tr>
<td><code>replicate</code></td>
<td>Int -&gt; a -&gt; [a]</td>
<td>Make a list of n copies of an item: replicate 3 'c' → &quot;ccc&quot;</td>
</tr>
<tr>
<td><code>take</code></td>
<td>Int -&gt; [a] -&gt; [a]</td>
<td>Take n elements from the front: take 3 &quot;Peccary&quot; → &quot;Pec&quot;</td>
</tr>
<tr>
<td><code>drop</code></td>
<td>Int -&gt; [a] -&gt; [a]</td>
<td>Drop n elements from the front: drop 3 &quot;Peccary&quot; → &quot;cary&quot;</td>
</tr>
<tr>
<td><code>splitAt</code></td>
<td>Int -&gt; [a] -&gt; ([a],[a])</td>
<td>Split at a given position: splitAt 3 &quot;Peccary&quot; → (&quot;Pec&quot;,&quot;cary&quot;)</td>
</tr>
<tr>
<td><code>reverse</code></td>
<td>[a] -&gt; [a]</td>
<td>Reverse the order of the elements: [2,1,3] → [3,1,2]</td>
</tr>
<tr>
<td><code>zip</code></td>
<td>[a] -&gt; [b] -&gt; [(a,b)]</td>
<td>Take a pair of lists into a list of pairs: zip [1,2] [3,4,5] → [(1,3),(2,4)]</td>
</tr>
<tr>
<td><code>unzip</code></td>
<td>[(a,b)] -&gt; ([a],[b])</td>
<td>Take a list of pairs into a pair of lists: unzip [(1,5),(3,6)] → ([1,3],[5,6])</td>
</tr>
</tbody>
</table>
concat

concat :: [[a]] -> [a]

Prelude> concat [[1,2,3],[4,5],[],[6,7,8]]
[1,2,3,4,5,6,7,8]

Prelude> concat ["hello"," ","world"]
"hello world"

Prelude> concat [[1,2,3],[False,True]]
—?????
### Monomorphc list functions in the Prelude

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>and</strong></td>
<td>[Bool] -&gt; Bool</td>
</tr>
<tr>
<td></td>
<td>The conjunction of a list of Booleans:</td>
</tr>
<tr>
<td></td>
<td>and [True, False] (\rightarrow) False</td>
</tr>
<tr>
<td><strong>or</strong></td>
<td>[Bool] -&gt; Bool</td>
</tr>
<tr>
<td></td>
<td>The disjunction of a list of Booleans:</td>
</tr>
<tr>
<td></td>
<td>and [True, False] (\rightarrow) True</td>
</tr>
</tbody>
</table>

There are also ad hoc polymorphic list functions in the Prelude, to be met later.
Caution: List functions in the Prelude

Some Prelude list functions do not actually look like

Prelude> :t length
length :: [a] -> Int

But rather

Prelude> :t length
length :: Foldable t => t a -> Int

For now, wherever you see Foldable, replace every t a by [a].