Lesson 3: Lambda Functions and Lexical Scope
Lambda functions

The general shape of a lambda function:

\( \lambda x \rightarrow x \)

The forward-slash \( \backslash \) is meant to remind you of the Greek lambda \( \lambda \). Argument(s) of the function:

Body of the lambda function: can be as long and complex as any other Haskell function.
Lambda functions

Typically, the lambda function

\( \lambda x \rightarrow x \)

is the **identity function** which returns the **value of its argument** as output:

ghci> (\x -> x) 4
4

ghci> (\x -> x) "hi"
"hi"

ghci> (\x -> x) [1,2,3]
[1,2,3]
Writing your own `where` clause

The original Haskell code

```haskell
sumSquareOrSquareSum x y = if sumSquare > squareSum
  then sumSquare
  else squareSum
  where sumSquare = x^2 + y^2
       squareSum = (x+y)^2
```

can be rewritten into the equivalent but hideous code:

```haskell
sumSquareOrSquareSum x y = if x^2 + y^2 > (x+y)^2
  then x^2 + y^2
  else (x+y)^2
```

**Exercise:** give two basic reasons why the code here is definitely ugly.
Writing your own \texttt{where} clause

Now, consider the \texttt{body} of the original function

\begin{verbatim}
body sumSquare squareSum = if sumSquare > squareSum
  then sumSquare
  else squareSum
\end{verbatim}

We can rewrite the function as the equivalent Haskell code:

\begin{verbatim}
sumSquareOrSquareSum x y = body (x^2 + y^2) ((x+y)^2)
\end{verbatim}

However, one main weakness of the approach is that we introduce the new intermediate Haskell function \texttt{body} although we use it only once and then get rid of it 😊
Writing your own `where` clause

The arguments are passed into the lambda function:

```
sumSquareOrSquareSum x y = (\sumSquare squareSum ->
                           if sumSquare > squareSum
                              then sumSquare
                              else squareSum) (x^2 + y^2) ((x+y)^2)
```

Body of the lambda function

In this way, every `where` clause of the Haskell code can be translated into a `lambda function` applied to an expression.
Same function using a \texttt{let} expression

The very same function

\texttt{sumSquareOrSquareSum}

can be also written using a \texttt{let} expression:

\begin{verbatim}
sumSquareOrSquareSum x y = let sumSquare = (x^2 + y^2)
                          squareSum = (x+y)^2
                in
                if sumSquare > squareSum
                   then sumSquare
                   else squareSum
\end{verbatim}
An exercise in lambdas

The `overwrite` function defined using let expressions

```plaintext
overwrite x = let x = 2
                in
                 let x = 3
                 in
                  let x = 4
                  in
                   x
```

is both correct and essentially useless! Can you guess what it does?

**Exercise.** Rewrite the function by using only lambda functions.
Lexical scope in Haskell

The *lexical scope* of a variable declaration

```
    let x = ... where x = ...
```

is the *area of the source code* in which the declaration applies.

```plaintext
overwrite x = let x = 2
   in
    let x = 3
   in
     let x = 4
   in
    x
```
Lexical scope in Haskell

The **lexical scope** of a variable declaration

```
let x = ...     where x = ...
```

is the **area of the source code** in which the declaration applies.
Lexical scope in Haskell

The **lexical scope** of a variable declaration

\[
\text{let } x = \ldots \quad \text{where } x = \ldots
\]

is the **area of the source code** in which the declaration applies.
Lexical scope in Haskell

The **lexical scope** of a variable declaration

```
let x = ... where x = ...
```

is the **area of the source code** in which the declaration applies.
Lexical scope in Haskell

\[ x = 4 \]

The variable looks at the top-level definition of the variable

\[ \text{add1 } y = y + x \]

The bound variable \( x \) looks for its definition and finds here the first lambda binder \( \lambda x \)

\[ \text{add2 } y = (\lambda x \rightarrow y + x) \, 3 \]

The bound variable \( y \) looks for its definition and finds here the first lambda binder \( \lambda y \)

\[ \text{add3 } y = (\lambda y \rightarrow (\lambda x \rightarrow y + x) \, 1) \, 2 \]
Lexical scope in Haskell

When we execute the three variants of the adder function:

```ghci
ghci> add1 1
5
ghci> add2 1
4
ghci> add3 1
3
```

we see how the value assigned to each variable \( x \) and \( y \) in the expression \( y + x \) depends on its lexical scope.
Discussion: lexical scope in JavaScript

JavaScript is another language with a strong support for lambda functions.

Typically, the Haskell function

\[ \lambda x \to x \]

would be created in the following way in JavaScript:

```javascript
function(x)
{
    return x;
}
```

One danger with JavaScript is to accidentally declare a global variable.
Discussion: lexical scope in JavaScript

Typically, the JavaScript function for addition is fine

```javascript
var libraryAdd = function(a,b){
    var c = a + b;
    return c;
}
```

but forgetting the `var` keyword in the body of the function

```javascript
var libraryAdd = function(a,b){
    c = a + b;
    return c;
}
```

has the undesired consequence of declaring the variable `c` as a **global variable** instead of as the intended **local variable**.
Discussion: lexical scope in JavaScript

The consequences of this slight mistake can be disastrous.

Indeed, consider the JavaScript code:

```javascript
var a = 2;
var b = 3;
var c = a + b;
var d = libraryAdd(10,20);
console.log(c);
```

The problem here is that the function call

```
libraryAdd(10,20)
```

will secretly store the value 30 in the global variable `c`.

As a result, the code will produce the value 30 and not 5 as expected 😞
Discussion: lexical scope in JavaScript

For that reason, JavaScript developers often use a pattern called *immediately invoked function expression* (IIFE) based on a lambda function:

```
(function(){
    var a = 2;
    var b = 3;
    var c = a + b;
    var d = libraryAdd(10,20);
    console.log(c);
})();
```

which delimits the **lexical scope** in which the variable `c` is defined.

As a result, the JavaScript function will produce the expected value 5 😊