Guards
In pattern matching, you have to specify exact patterns and values to distinguish different cases. If you need to check inequalities or call functions in order to make a match, you can use guards instead:

```haskell
iqGuards :: Int -> [Char]
iqGuards n
    | n > 150 = "amazing!"
    | n > 100 = "cool!"
    | otherwise = "oh well..."
```

Recursion
Since variables in Haskell are immutable, our only way of achieving iteration is through recursion. For example, the reverse function receives a list as its input and outputs the same list but with its elements in reverse order:

```haskell
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

Currying
As you know, you can turn any infix operator into a prefix operator by putting it in parentheses:
```
(+) 3 4
7
```
Now currying allows us to place the parentheses differently:
```
(+ 3) 4
7
```
By “fixing” the first input to (+) to be 3, we created a new function (+ 3) that receives only one (further) input.

Lambda Expressions
Lambda expressions are anonymous functions that are useful whenever we need a simple helper function. Examples for lambda expressions:
```
zipWith (\x y -> x^2 + y^2) [1..10] [11..20]
[122,148,178,212,250,292,338,388,442,500]
map (\x -> (x, x^2, x^3)) [1..5]
[(1, 1, 1),(2, 4, 8),(3, 9, 27),(4, 16, 64),(5, 25, 125)]
```
The $ Operator

The $ operator is defined as follows:

\[ f \, $ \, x = f \, x \]

It has the lowest precedence, and therefore, the value on its right is evaluated first before the function on its left is applied to it. As a consequence, it allows us to omit parentheses:

\[
\text{negate (sum (map sqrt [1..10]))}
\]

can be written as:

\[
\text{negate \, $ \, sum \, $ \, map \, sqrt \, [1..10]}
\]

Function Composition

Similarly, we can use function composition to make our code more readable and to create new functions. As you know, in mathematics, function composition works like this:

\[ (f \circ g) (x) = f(g(x)) \]

In Haskell, we use the "." character instead:

\[
\text{map \, (\lambda \, xs \rightarrow \text{negate \, (sum \, (tail \, xs))}) \}}
\]

\[ \text{[[1..5],[3..6],[1..7]]} \]

Can be written as:

\[
\text{map \, (negate \, \cdot \, \text{sum} \, \cdot \, \text{tail}) \}}
\]

\[ \text{[[1..5],[3..6],[1..7]]} \]

Data Types

Data types can be declared as follows:

\[
data \, \text{Bool} = \text{False} \mid \text{True}
data \, \text{Shape} = \text{Circle} \, \text{Float} \, \text{Float} \, \text{Float} \mid 
\text{Rectangle} \, \text{Float} \, \text{Float} \, \text{Float} \, \text{Float}
deriving \, \text{Show}
\]

Then we can construct values of these types like this:

\[ x = \text{Circle} \, 3 \, 4 \, 5 \]

The "deriving Show" line makes these values printable by simply using the show function (object-to-string conversion) the way it is defined for the individual objects (here: floats).

Data Types

We can use pattern matching on our custom data types:

\[
\text{surface} :: \text{Shape} \rightarrow \text{Float}
\]

\[
\text{surface} \, (\text{Circle} \, _ \, _ \, r) = 3.1416 \times r^2
\]

\[
\text{surface} \, (\text{Rectangle} \, x1 \, y1 \, x2 \, y2) = (\text{abs} \, x2 - x1) \times (\text{abs} \, y2 - y1)
\]

\[
\text{surface} \, $ \, \text{Circle} \, 10 \, 20 \, 10
\]

314.16

Records

If we want to name the components of our data types, we can use records:

\[
data \, \text{Car} = \text{Car} \, \{\text{company} :: [\text{Char}], \text{model} :: [\text{Char}], \text{year} :: \text{Int}\} \, \text{deriving Show}
\]

\[
\text{myCar} = \text{Car} \, \{\text{company}=\text{"Ford"}, \text{model}=\text{"Mustang"}, \text{year}=1967\}
\]

\[
\text{company} \, \text{myCar} \,
\text{"Ford"}
\]

Input/Output with “do”

Purely functional code cannot perform user interactions such as input and output, because it would involve side effects. Therefore, we sometimes have to use impure functional code, which needs to be separated from the purely functional code in order to keep it (relatively) bug-safe. In Haskell, this is done by so-called Monads. To fully understand this concept, more in-depth study is necessary.

However, in this course, we do not need to perform much input and output. We can use a simple wrapper (or "syntactic sugar") for this – the "do" notation.
Input/Output with "do"
In a do-block, we can only use statements whose type is "tagged" IO so that they cannot be mixed with purely functional statements.
Example for a program performing input and output:
```haskell
main = do
    putStrLn "Hello, what’s your name?"
    name <- getLine
    putStrLn ("Hey " ++ name ++ ", you rock!")
```

The Syntax of $\mathcal{L}$
Now back to our programming language $\mathcal{L}$...

A computation of a program $\varphi$ is defined to be a sequence $s_1, s_2, \ldots, s_k$ of snapshots of $\varphi$ such that $s_k$ is the successor of $s_i$ for $i = 1, 2, \ldots, k - 1$ and $s_k$ is terminal.

Computable Functions
What does it exactly mean when we say that a program computes a function?
We would like to find a precise definition for this.
Let $\varphi$ be any program in the language $\mathcal{L}$ and let $r_1, \ldots, r_m$ be $m$ given numbers.
We form the state $\sigma$ of $\varphi$ which consists of the equations $X_1 = r_1, X_2 = r_2, \ldots, X_m = r_m, Y = 0$

$t_e$ together with the equations $V = 0$ for all other variables $V$ in $\varphi$.
We call this the initial state.

Moreover, we call the snapshot $(1, \sigma)$ the initial snapshot.
When running the program, there are two possible outcomes:

Case 1:
There is a computation $s_1, s_2, \ldots, s_k$ of $\varphi$ beginning with the initial snapshot.
Then we write $\psi_\varphi(r_1, r_2, \ldots, r_m)$ for the value of the variable $Y$ at the terminal snapshot $s_k$.

Case 2:
There is no such computation; i.e., there is an infinite sequence $s_1, s_2, s_3, \ldots$ beginning with the initial snapshot.
In this case, $\psi_\varphi(r_1, r_2, \ldots, r_m)$ is undefined.

Sample Computation
```
[A] IF X=0 GOTO B (1) (1, (X = r, Y = 0, Z = 0)),
Z ← Z+1 (2) (4, (X = r, Y = 0, Z = 0)),
IF Z=0 GOTO E (3) (5, (X = r - 1, Y = 0, Z = 0)),
[B] X ← X-1 (4) (6, (X = r - 1, Y = 1, Z = 0)),
Y ← Y+1 (5) (7, (X = r - 1, Y = 1, Z = 1)),
Z ← Z+1 (6) (1, (X = r - 1, Y = 1, Z = 1)),
IF Z=0 GOTO A (7)
```

We write:
```
(1, (X = r, Y = r, Z = r)),
(2, (X = r, Y = r, Z = r)),
(3, (X = r, Y = r, Z = r + 1)),
(4, (X = r, Y = r, Z = r + 1))
```
Computation

For any program \( P \) and any positive integer \( m \), the function \( \psi_P^m(r_1, r_2, \ldots, r_m) \) is said to be computed by \( P \).

We allow any program to be run with any number of inputs.

Consider the summation program from the previous lecture:

\[
\psi_P^2(r_1, r_2) = r_1 + r_2 \\
\psi_P^1(r_1) = r_1 + 0 = r_1 \\
\psi_P^3(r_1, r_2, r_3) = r_1 + r_2
\]