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:

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

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:

```
negate $ sum $ map sqrt [1..10]
```

can be written as:

```
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 ° g) (x) = f(g(x))
```

In Haskell, we use the “.” character instead:

```
map (\xs -> negate (sum (tail xs)))
[[1..5],[3..6],[1..7]]
```

Can be written as:

```
map (negate . sum . tail)
[[1..5],[3..6],[1..7]]
```

Data Types
Data types can be declared as follows:

```
data Shape = Circle Float Float Float 
             | Rectangle Float Float Float Float 
             deriving Show
```

Then we can construct values of these types like this:

```
x = 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).

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

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

```
myCar = Car {company="Ford", model="Mustang", year=1967}
```

```
company myCar
"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.

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 \( s_{i+1} \) is the successor of \( s_i \) for \( i = 1, 2, \ldots, k - 1 \) and \( s_k \) is terminal.

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 \( \psi^{(m)}(r_1, r_2, \ldots, r_m) \) 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, \( \psi^{(m)}(r_1, r_2, \ldots, r_m) \) is undefined.

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
\( X_1 \equiv r_1, X_2 \equiv r_2, \ldots, X_m \equiv r_m, Y \equiv 0 \)
together with the equations \( V \equiv 0 \) for all other variables \( V \) in \( \varphi \).
We call this the initial state.

Moreover, we call the computation \((1, \sigma)\) 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 \( \psi^{(m)}(r_1, r_2, \ldots, r_m) \) 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, \( \psi^{(m)}(r_1, r_2, \ldots, r_m) \) is undefined.
Sample Computation

[A] IF $X = 0$ GOTO B (1)
\[ (X = r, Y = 0, Z = 0) \]
$Z \leftarrow Z + 1$ (2)
\[ (X = r, Y = 0, Z = 0) \]
IF $Z = 0$ GOTO E (3)
\[ (X = r - 1, Y = 0, Z = 0) \]
[B] $X \leftarrow X - 1$ (4)
\[ (X = r - 1, Y = 1, Z = 0) \]
$Y \leftarrow Y + 1$ (5)
\[ (X = r - 1, Y = 1, Z = 1) \]
$Z \leftarrow Z + 1$ (6)
\[ (X = r - 1, Y = 1, Z = 1) \]
IF $Z = 0$ GOTO A (7)
\[ (X = 0, Y = r, Z = r) \]
We write:
\[
\begin{align*}
[A] &\quad (1, (X = r, Y = 0, Z = 0)) \\
&\quad (2, (X = r, Y = 0, Z = 0)) \\
&\quad (3, (X = r - 1, Y = 0, Z = 0)) \\
&\quad (4, (X = r - 1, Y = 1, Z = 0)) \\
&\quad (5, (X = r - 1, Y = 1, Z = 1)) \\
&\quad (6, (X = r - 1, Y = 1, Z = 1)) \\
&\quad (7, (X = 0, Y = r, Z = r + 1)) \\
&\quad (8, (X = 0, Y = r, Z = r + 1))
\end{align*}
\]

Computation

For any program $\varphi$ and any positive integer $m$, the function $\psi_{\varphi}^m(r_1, r_2, \ldots, r_m)$ is said to be computed by $\varphi$.

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

Consider the summation program from the previous lecture:
\[
\begin{align*}
\psi_{\varphi}^1(f_1, r_2) &= r_1 + r_2 \\
\psi_{\varphi}^2(r_1) &= r_1 + 0 = r_1 \\
\psi_{\varphi}^3(r_1, r_2, r_3) &= r_1 + r_2
\end{align*}
\]

In general, a partial function $f$ on a set $S^n$ is a function whose domain is a subset of $S^n$. If a partial function on $S^n$ has the domain $S^n$, then it is called total.

Computation

A given partial function $g$ (of one or more variables) is said to be partially computable if it is computed by some program.

This is the case if there is a program $\varphi$ such that $g(r_1, r_2, \ldots, r_m) = \psi_{\varphi}^m(r_1, r_2, \ldots, r_m)$ for all $r_1, r_2, \ldots, r_m$.

This means not only that both sides have the same value when they are defined, but also that when either side of the equation is undefined, the other one is as well.

A function is said to be computable if it is both partially computable and total.

Macros

So far we have used macros in an informal way. Let us now develop a more precise definition of them.

Let $f(x_1, x_2, \ldots, x_n)$ be a partially computable function, and $\varphi$ be a program that computes $f$.

Let us assume that
- the variables in $\varphi$ are named $Y, X_1, \ldots, X_n, Z_1, \ldots, Z_k$,
- the labels in $\varphi$ are named $E, A_1, \ldots, A_l$, and
- for each instruction of $\varphi$ of the form IF $V = 0$ GOTO $A_i$ there is in $\varphi$ an instruction labeled $A_i$.

We can modify any program of the language $L$ to comply with these assumptions.

We write:
\[ \varphi = \varphi(Y, X_1, \ldots, X_n, Z_1, \ldots, Z_k; E, A_1, \ldots, A_l) \]

In particular, we will use:
\[ Q_m = \varphi(Z_m, Z_{m+1}, \ldots, Z_{m+n}, Z_{m+n+1}, \ldots, Z_{m+n+k}; E_m, A_{m+1}, \ldots, A_{m+k}) \]

for a given value $m$. 

Now we want to use macros of the form:

$$W \leftarrow f(V_1, \ldots, V_n)$$

in our programs, where $W$, $V_1$, $\ldots$, $V_n$ can be any variables; $W$ could be among $V_1$, $\ldots$, $V_n$.

We expand the macro as follows:

\[
\begin{align*}
Z_m & \leftarrow 0 \\
Z_{m+1} & \leftarrow V_1 \\
Z_{m+2} & \leftarrow V_2 \\
& \quad \vdots \\
Z_{m+n} & \leftarrow V_n \\
Z_{m+n+1} & \leftarrow 0 \\
Z_{m+n+2} & \leftarrow 0 \\
& \quad \vdots \\
Z_{m+n+k} & \leftarrow 0 \\
\end{align*}
\]

$$Q_m \leftarrow [E_m] W \leftarrow Z_m$$

Whenever we expand a macro, the number $m$ has to be chosen large enough so that none of the variables or labels in $Q_m$ occur in the main program.

Note that in the expansion the output and local variables are set to zero, although at the start of the main program they are set to zero anyway. This is necessary because the macro expansion may be part of a loop in the main program.