Two tasks have to be solved to implement Haskell via the Lambda Calculus:

1. Define evaluation strategy for $\rightarrow_{\beta_0}$ that corresponds to the evaluation strategy of Haskell.

2. Define (automatic) translation from (simple) Haskell to Lambda Terms.

$\rightarrow_{\beta_0}$ is confluent, but already $\rightarrow_{\beta}$ is not terminating: (Slide 56)

\[(\lambda x. xx) \ (\lambda x. xx)\]

$\Rightarrow (xx) \ [x/\lambda x. xx]$

$= (\lambda x. xx) \ (\lambda x. xx)$

$\Rightarrow_{\beta} \ldots$

It can depend on the evaluation strategy whether one terminates or not:

\[(\lambda x. y) \ (\lambda x. xx) \ (\lambda x. xx)\]

$\Rightarrow_{\beta} \ \ y$

$\Rightarrow_{\beta} \ \ (\lambda x. y) \ ((\lambda x. xx) \ (\lambda x. xx))$

$\Rightarrow_{\beta} \ y \ldots$
Here, outermost evaluation terminates, but innermost evaluation does not.

In order to correspond to Haskell, we have to use $\rightarrow_{\beta}$ with leftmost outermost evaluation strategy.

Moreover, we should not always evaluate until one reaches a normal form:

In a term $t_1 \ldots t_n$ where $t_n$ can never be evaluated further, one should also not evaluate $t_1, \ldots, t_{n-1}$ further, even if they are not yet in normal form.

Three cases:

1. **Top symbol is a data constructor**

   \[(+2) : C 3 \quad (\cdot) ((+) 12) \in 3\]

   is not evaluated further although there is a $\delta$-rule $1+2 \rightarrow 3$

   **Reason:** Top symbol is $: C$ and there is no $\delta$-rule for $: C$ (i.e., $: C$ is a data constructor)

2. **Top symbol is a variable**

   \[x \ (1+2)\]

   Here, $1+2$ is also not evaluated further, because there is no rule to evaluate the application of the variable $x$.

3. **Top symbol is $\lambda$**
- top symbol is $\lambda$

$$\lambda x_\vdash (1+2)$$

Again, $1+2$ is not evaluated further.

**Def. 3.3.1 (Weak Head Normal Form, WHNF)**

A term $t$ is in WHNF if it is a normal form $\Rightarrow^\vdash$ or it has one of the following forms:

- $c \ t_1 \ldots t_n$ for $t_1, \ldots, t_n \in A$, $c \in \mathcal{C}$ such that there are no rules for $c$ in $\delta$ (i.e., $c$ is a construct)
- $x \ t_1 \ldots t_n$ for $x \in \mathcal{V}$, $t_1, \ldots, t_n \in A$
- $\lambda \ x \cdot t$ for $x \in \mathcal{V}$, $t \in A$

**Def. 3.3.2 (Weak Head Normal Order Reduction)**

(WHNO)

The WHNO-reduction on Lambda Terms is defined as follows:

$t \rightarrow^r \ r$ if $t$ is not in WHNF and

$t \rightarrow^\vdash_p \ r$ using the leftmost outermost strategy.

It remains to solve Task 2, translate Simple Haskell expressions to Lambda Terms.  
(Slide 57)
\[ \text{Lam} : \text{Exp} \rightarrow \Lambda \]

Set of \textbf{Lambda Terms} over the constants:
- \( \text{C}_0 \) (pre-defined func. symbols like +, not, sqrt, ...,
- \( \text{isa}, \text{sel}_i \), ...

\( \cup \text{Con} \) (data constructors including
- the ones Int, Float, Char, Bool, ...)

1. \( \text{Lam} (\text{var}) = \text{var} \)
2. \( \text{Lam} (c) = c \), where \( c \in \text{C}_0 \cup \text{Con} \)
3. \( \text{Lam} ((\text{exp}_1, \ldots, \text{exp}_n)) = \text{tuple}_n \cdot \text{Lam} (\text{exp}_1) \ldots \text{Lam} (\text{exp}_n) \) for \( n \in \{0,2,3,\ldots\} \)
4. \( \text{Lam} (\text{if} \; \text{exp}_1 \; \text{then} \; \text{exp}_2 \; \text{else} \; \text{exp}_3) = \text{if} \; \text{Lam} (\text{exp}_1) \; \text{Lam} (\text{exp}_2) \; \text{Lam} (\text{exp}_3) \)
5. \( \text{Lam} (\lambda \text{var} \rightarrow \text{exp}) = \lambda \text{var}. \text{Lam} (\text{exp}) \)

It remains to define \( \text{Lam} (\text{let} \; \text{var} = \text{exp} \; \text{in} \; \text{exp}') \).

**Case 1:** \( \text{var} \) does not occur free in \( \text{exp} \),
i.e., \( \text{var} = \text{exp} \) is not a recursive declaration

Then: \( \text{Lam} (\text{let} \; \text{var} = \text{exp} \; \text{in} \; \text{exp}') = \text{Lam} (\text{exp}') [\text{var} / \text{Lam} (\text{exp})] \)

Ex: \( \text{Lam} (\text{let} \; x = 3 \; \text{in} \; x + 2) \)

\( \Lambda \{ - \} \rightarrow \Lambda \)
\[ \text{ex. } \lambda x : \text{Nat} \ x = 0 \ \text{in } x + 1 \]
\[
= \underbrace{\lambda x : \text{Nat} (x + 2)}_{\text{(x+2)}} \ \underbrace{[x / \lambda x : \text{Nat} (3)]}_{3}
\]
\[
= 3 + 2
\]

Case 2: \( \text{Var} \) occurs free in \( \text{exp} \), i.e., \( \text{Var} = \text{exp} \) is a recursive declaration.

Let \( \text{fact} = \lambda x : \text{Nat} \ x = 0 \ \text{then} \ 1 \ \text{else} \ \text{fact}(x-1) \times x \) in fact 2.

According to the semantics of Haskell, \( \text{fact} \) should be the least fixpoint of the function that is computed by
the following Haskell expression:

\[
\lambda x : \text{Nat} \ x = 0 \ \text{then} \ 1 \ \text{else} \ \lambda x : \text{Nat} \ \text{fact}(x-1) \times x
\]

We add another constant fix to the Lambda Calculus which computes the least fixpoint of its argument.
Then the original (recursive) Haskell expression could be reformulated to the following non-recursive expression:

Let \( \text{fact} = \text{fix} (\lambda x : \text{Nat} \ x = 0 \ \text{then} \ 1 \ \text{else} \ \text{fact}(x-1) \times x) \) in fact 2.

This expression can be translated as in Case 1:

\[
\lambda x : \text{Nat} \ \text{fact} \ (\text{fix} (\lambda x : \text{Nat} \ x = 0 \ \text{then} \ 1 \ \text{else} \ \text{fact}(x-1) \times x))
\]
\[
= \text{fix} (\lambda x : \text{Nat} \ x = 0 \ \text{then} \ 1 \ \text{else} \ \text{fact}(x-1) \times x)
\]
General Translation of let-expressions:
\[
\text{Lam} (\text{let } \text{var} = \text{exp} \text{ in } \text{exp}') = \\
\text{Lam} (\text{exp}') \left[ \text{var}/(\text{fix} \ (\lambda \text{var} . \text{Lam} (\text{exp}))) \right]
\]

In order to evaluate Lambda Terms that are built with fix, we need an appropriate \( \vdash \) rule such that:
\[
\frac{}{\text{fix} \ t \ \rightarrow^* \ t \ (\text{fix} \ t) \ \rightarrow^* \ t \ (t \ (\text{fix} \ t)) \rightarrow^* \ldots}
\]
\[
\equiv \ \\ ^* \ \\ ^* \ \\ ^* \ \\
\equiv 1 \ \\
\equiv t(1) \ \\
\equiv t(t(1))
\]

So in this way, a function \( t \) can be unfolded arbitrary many times.

Def of \( \text{Lam} \) also works for non-recursive declarations:
\[
\text{Lam} (\text{let } x = 3 \text{ in } x + 2) = \\
\text{Lam} (x + 2) \left[ x / \text{fix} \ (\lambda x . \text{Lam} (3)) \right] \\
= (\text{fix} \ (\lambda x . 3)) + 2 \\
\rightarrow^* \ (\lambda x . 3)(\text{fix} \ (\lambda x . 3)) + 2 \\
\rightarrow^* 3 + 2
\]

Def 3.3.3 (Translation of Simple Haskell to)
Def 3.3.3 (Translation of Simple Haskell to Lambda Terms)

Law: \( \text{Exp} \rightarrow A \) is defined on Slide 57.

To evaluate a complex Haskell expression \( \text{exp} \) in a complex \( \text{H} \)-program, where \( P \) is the sequence of function and pattern declarations:

- We evaluate \( \text{Law} \left( \left( \text{let} \ P \ \text{in} \ \text{exp} \right) \right) \)

using \( \text{WHNO} \)-reduction.

Transformation from Sect. 2.2.3 in order to transform complex to Simple Haskell.

Remaining question:

Which \( \delta \)-rules are used in this reduction?