Now we want to define a mapping Val from Haskell-expressions to mathematical objects from the set Dom. To this end, we also want to define the semantics of sub-expressions. However, sub-expressions may contain variables that were defined in the context of this sub-expression.

```
let x = 3 in plus x 2
```

To compute the semantics of this sub-expression, one needs the information of the context that x is 3 and plus might also have been defined previously.

One can only define the semantics of an expression wrt an environment that assigns values to all variables that occur (free) in this expression.

An environment $p$ is a function that maps variables to elements of $Dom$.

If $p(x) = 2$ and $p(plus)$ is the addition function then the semantics of

```
plus x 3
```

is 5.

So our semantics is a function $Val: Exp \rightarrow Env \rightarrow Dom$. 

FP 2016 Seite 1
Def 2.2.4 (Environment) (Slide 37)

For a \( H \)-program with the domain \( \text{Dom} \), an \( \text{environment} \) is a partial function mapping variables to \( \text{Dom} \), which is only defined for finitely many variables: \( \rho : \text{Var} \rightarrow \text{Dom} \). Let \( \text{Env} \) denote the set of all environments. An environment \( \rho \) that is only defined on \( \text{var}_1, \ldots, \text{var}_n \) where \( \rho(\text{var}_i) = d_i \) is also denoted \( \rho = \{ \text{var}_1, \ldots, \text{var}_n / d_1, \ldots, d_n \} \).

We write \( \rho_1 + \rho_2 \) for the environment that is like \( \rho_2 \) whenever \( \rho_2 \) is defined. Otherwise, it is like \( \rho_1 \):

\[
(\rho_1 + \rho_2) (\text{var}) = \begin{cases} 
\rho_2 (\text{var}), & \text{if } \rho_2 (\text{var}) \text{ is defined} \\
\rho_1 (\text{var}), & \text{otherwise}
\end{cases}
\]

Let \( \text{env} \) be the initial environment which assigns the correct meaning to all pre-defined operations in Haskell. E.g.: \( (\text{co}(+) \times y = \begin{cases} x + y, & \text{if } x, y \in \mathbb{Z} \text{ or } x, y \notin \mathbb{N} \\
1, & \text{otherwise}
\end{cases} \)

Semantics of an expression \( \text{exp} \) in an environment \( \rho \) will be denoted \( \text{Val} [\text{exp}] \_\rho \)

\( \text{Val} : \text{Exp} \rightarrow \text{Env} \rightarrow \text{Dom} \)

To simplify the definition of \( \text{Val} \), we only regard a subset of Haskell without pattern matching: “Simple
Haskell". In Sect 2.2.3, we will show that every complex H-program can be automatically transformed into a simple H-program.

Restrictions of Simple Haskell:

0. No type synonyms, no type classes
1. Just one declaration, which must be a pattern declaration of the form: var = exp
2. No pre-defined lists.
3. No applications of the form \((\text{expr}_1 \text{expr}_2 \ldots \text{expr}_n)\) for \(n > 2\). But: \((\text{expr}_1 \text{expr}_2) \text{expr}_3 \ldots \) \text{expr}_n is allowed.
4. No "case" construct.
5. Only lambda-expressions of the form \(\backslash \text{var} \rightarrow \text{exp}\)
6. No "where" (but "let" is allowed).

Def 2.2.5 (Simple Haskell)

see slide 38

Ex. for a simple Haskell program:

\[
\text{fact} = \backslash x \rightarrow \text{if } x \leq 0 \text{ then } 1 \text{ else } \text{fact} (x-1) \times x
\]

This simple Haskell program contains the pre-defined variables \(\leq, -, \times\) whose semantics is already defined in the initial environment \(\text{co}\).

Instead of determining the semantics of \(\text{exp}'\),
the program \( \texttt{var} = \texttt{exp} \), we can determine
the semantics of
\[
\texttt{let} \ \texttt{var} = \texttt{exp} \ \texttt{in} \ \texttt{exp}'
\]
in the empty program.

E.g.: If we would like to know the semantics of \texttt{fact 2}
in the \texttt{fact-program}, we can instead compute
the semantics of the following expression in the
empty program:

\[
\texttt{let fact} = \langle \texttt{x} \mapsto \ldots \rangle \ \texttt{in} \ \texttt{fact 2}
\]

Goal: Define semantics of expressions \texttt{exp}
Val \( \Gamma \ \texttt{exp} \ \Sigma \) where \( \Sigma \) is an environment
that defines the meaning of all variables that
occur \texttt{free} in \( \texttt{exp} \). (Moreover, \( \texttt{exp} \) must be
syntactically correct and well typed.)

A variable occurring in an expression is \texttt{free} unless it
is bounded by a "let" or a lambda ("\( \lambda \)").

**Def 2.26. (Free variables of H-expressions)**
For every simple H-expression \( \texttt{exp} \), we define \texttt{free(}\texttt{exp})
(the set of its \texttt{free} variables) as follows:

- \( \texttt{free(var)} = \{ \texttt{var} \} \)
- \( \texttt{free(constr)} = \texttt{free(integer)} = \texttt{free(float)} = \texttt{free(char)} = \emptyset \)
Now we can define \( \text{Val} \ II \exp II \varphi \) where \( \varphi \) must be defined on all variables in \( \text{free} \( \exp \) \). (Slide 33+40)

1. \( \exp \) is a variable

\[
\text{Val} \ II \ var II \varphi = \varphi(var)
\]

Ex: If \( \varphi(x) = 2 \), \( \varphi(\text{plus}) \) is the addition fact, then:
\[
\text{Val} \ II \times II \varphi = \varphi(x) = 2 \text{ in Constructions, in Dom}
\]
\[
\text{Val} \ II \text{ plus II } \varphi = \varphi(\text{plus}) = \text{add, fact in Functions in Dom}
\]

2. \( \exp \) is a constructor \( \text{constro} \) from \( \text{Con}_0 \)

\[
\text{Val} \ II \text{ constro II } \varphi = \text{constro in Constructions, in Dom}
\]

Ex: \( \text{Val} \ II 5 II \varphi = 5 \) in Constructions, in Dom

3. \( \exp \) is a constructor \( \text{constn} \in \text{Con}_n \), where \( n > 0 \)

\[
\text{Val} II \text{ constn II } \varphi = f \text{ in Functions in Dom}
\]
where \( f, d_1, \ldots, d_n = (\text{constn}, d_1, \ldots, d_n) \) in Constructions, in Dom

Ex: data Nats = Zero | Succ Nats
Val II Succ II \sigma = f_{\text{succ}}
where \( f, d = (\text{Succ}, d) \) in \text{Constructions, in Dom}

4. exp is a tuple \((\text{exp}_1, \ldots, \text{exp}_n), n \in \{0, 1, 2, 3, 4, \ldots\}\)
Val II (\text{exp}_1, \ldots, \text{exp}_n) II \sigma = (\text{Val II exp}_1 II \sigma, \ldots, \text{Val II exp}_n II \sigma)
in \text{Tuples}_n \text{ in Dom}

5. exp is a one-component tuple
Val II (\text{exp}) II \sigma = \text{Val II exp} II \sigma

6. exp is an application \((\text{exp}_1 \text{ exp}_2)\)
Val II (\text{exp}_1 \text{ exp}_2) II \sigma = f (\text{Val II exp}_2 II \sigma)
where \text{Val II exp}_1 II \sigma = f \text{ in Functions in Dom}

\text{Ex: Val II Succ Zero II} \sigma = f_{\text{succ}} (\underbrace{\text{Val II Zero II} \sigma}_\text{Zero})
= (\text{Succ, Zero}) \text{ in Constructions in Dom}

7. exp is if \text{exp}_1 \text{ then } \text{exp}_2 \text{ else } \text{exp}_3
Val II \text{ if } \text{exp}_1 \text{ then } \text{exp}_2 \text{ else } \text{exp}_3 II \sigma = \begin{cases} \text{Val II exp}_1 II \sigma, & \text{if Val II exp}_1 II \sigma = \text{True} \\ \text{Val II exp}_2 II \sigma, & \text{otherwise} \end{cases}
in \text{Constructions in Dom}

8. exp is an let-expression
What is \( \text{Val} \Rightarrow \text{let } \text{var} = \text{exp} \text{ in } \text{exp}' \Rightarrow \text{if } \) \\

**Case 1:** \text{var} does not occur in \( \text{exp} \) (no recursion) \\
\( \text{Val} \Rightarrow \text{let } x = 3 \text{ in } \text{plus} \times y \Rightarrow \text{if } \) \\
\( \text{Val} \Rightarrow \text{plus} 	imes y \Rightarrow (p + \{x/3\}) = \) \\
\( \text{Val} \Rightarrow \text{plus} = 3 \) \\
\( \text{Val} \Rightarrow \text{if } \) \\
\( \text{Val} \Rightarrow \text{exp} \) \\
\( \text{Val} \Rightarrow \text{exp}' \Rightarrow \text{if } \) \\
\( \text{Val} \Rightarrow \text{exp} \) \\
where \\
\( p(y) = 2 \) \\
\( s(\text{plus}) = \text{addition} \) \\
\( s(\text{plus}) = 3 \) \\
\( s(x) = 5 \) \\

This does not work if \( \text{var} \in \text{free}(\text{exp}) \).

**Case 2:** \text{var} \in \text{free}(\text{exp}) \\
\( \text{Val} \Rightarrow \text{let } \text{fact} = \{x \rightarrow \text{if } x < 0 \text{ then } \text{elsefact}(x - 1) \times x \} \text{ in } \text{fact} \Rightarrow \text{if } \) \\
\( \text{Val} \Rightarrow \text{fact} 2 \Rightarrow \)  \\
where \( p' \) is like \( p \) for all variables except \( \text{fact} \).

The meaning of \( \text{fact} \) should be the least fixpoint of the following higher-order function \( \text{ff} \): \\
\( \text{ff}(d) = \text{Val} \Rightarrow \text{\{x \rightarrow if } x < 0 \text{ then } \text{elsefact}(x - 1) \times x \} \Rightarrow (p + \{\text{fact} / d\}) \) \\
\( s; \) \\
\( \text{Val} \Rightarrow \text{let } \text{fact} = \{x \rightarrow \ldots \text{fact}(x - 1) \times x \} \text{ in } \text{fact} 2 \Rightarrow \text{if } \) \\
\( \text{Val} \Rightarrow \text{fact} 2 \Rightarrow (p + \{\text{fact} / \text{lfp \text{ff} d}\}) \) \\
where \( \text{ff}(d) = \text{Val} \Rightarrow \text{\{x \rightarrow if } x < 0 \text{ then } \text{elsefact}(x - 1) \times x \} \Rightarrow (p + \{\text{fact} / d\}) \)
In general:

\[ \text{Val II } \text{let } \text{var } = \text{exp in } \text{exp' II } s = \text{Val II } \text{exp' II } (s + \{ \text{var } / \text{exp' II } f \}) \]

where \( f (d) = \text{Val II } \text{exp' II } (s + \{ \text{var } / d \}) \)

This definition can be used for both recursive and non-recursive declarations.

If \( \text{var } \notin \text{free} (\text{exp}) \), then

\[ f (d) = \text{Val II } \text{exp' II } (s + \{ \text{var } / d \}) = \text{Val II } \text{exp' II } s \]

So the only fixed point of \( f \) is \( \text{Val II } \text{exp' II } s \).

9. \text{exp is a lambda expression}

\[ \text{Val II } \backslash \text{var } \rightarrow \text{exp' II } s = f \text{ in Functions in } \text{Dom} \]

where \( f (d) = \text{Val II } \text{exp' II } (s + \{ \text{var } / d \}) \)

Ex: \( \text{Val II } \backslash x \rightarrow \text{plus } x \cdot y \text{ II } s = f \)

where \( f (d) = \text{Val II } \text{plus } x \cdot y \text{ II } (s + \{ x / d \}) \)

\[ = s (\text{plus}) \cdot d \cdot s (y) \]

\[ = d + 2 \]

Def 227 (Semantics of Simple Haskell-Programs)

Let \( \text{Dom} \) be the domain of a simple H-program, let \( \text{var } = \text{exp } \) be the (only) pattern declaration of the program, let \( \text{Exp} \) be the set of all simple H-expressions.

Then we define the function \( \text{Val } : \text{Exp } \rightarrow \text{Env } \rightarrow \text{Dom} \). 

See Slide 38+40.

The semantics of an expression \( \text{exp} \) that does not contain any free variables except \( \text{var} \) and pre-defined variables of Haskell is defined as:

\[
\text{val \_let \ \text{var} = \text{exp} \ \text{in} \ \text{exp} \ \text{II} \ \text{co}
\]