Expressions = central concept of functional programming
Every expression has a type. Interpreter first checks whether expression is well typed. If yes, then the expression is evaluated afterwards.

\[ t \quad \text{just computes the type of } \exp \text{ in } \text{GHC} \]

We now introduce the different forms of expressions (with their type and the value that it evaluates to).

- **Var** (strings with lower-case symbol)
- **Const** (data constructors, strings with upper-case symbol)
  - Are not evaluated, but used to represent objects of data types (e.g., True, False, [], ::, ...)
- **Integer** (0, 1, -1, ...) have type Int
- **Float** (-2.5, 3.4 e+23, ...) have the type Float
- **Char** (a, ..., z, A, ..., Z, '0', ..., '9', ' ', '\n', ..., newline) have the type Char
- \[ [\exp_1, \ldots, \exp_n] \] stands for the list of the expressions \exp_1, ..., \exp_n, i.e., for:
\[ \exp_1 : \exp_2 : \ldots : \exp_n : [\] \]
\exp_1, ..., \exp_n must all have the same type \( T \).
then \([\text{exp}_1, \ldots, \text{exp}_n]\) has type \([T]\)

\([1,2,3]\) has type \([\text{Int}]\)

\([\text{True}, \text{True}, \text{False}]\) has type \([\text{Bool}]\)

\([\text{square}, +2]\) has type \([\text{Int} \rightarrow \text{Int}]\)

\(\text{Int} \rightarrow \text{Int} \quad \text{Int} \rightarrow \text{Int}\)

\([1,2,3], [1,2,3], [1,2,3]\) has type \([[[\text{Int}]]]\)

\([1, \text{True}]\) is not well typed

- **String**: List of characters, i.e., expression of type

  \([\text{Char}] = \text{String}\).

\(['h', 'e', 'l', 'l', 'o'] = "hello"\)

- \((\text{exp}_1, \ldots, \text{exp}_n)\), where \(n \geq 0\)

  stands for the tuple of \(n\) expressions.

  If \(\text{exp}_1\) has type \(T_1\), \ldots, \(\text{exp}_n\) has type \(T_n\),

  then \((\text{exp}_1, \ldots, \text{exp}_n)\) has type \((T_1, \ldots, T_n)\).

\((1, \text{True}, [1,2,3])\) has type \((\text{Int}, \text{Bool}, [\text{Int}]\))

- \((\text{exp}_1 \ldots \text{exp}_n)\), \(n \geq 2\)

  means that: \(\ldots ((\text{exp}_1 \text{exp}_2) \text{exp}_3) \ldots \ldots \text{exp}_n)\)

  i.e.: function application
plus 2 3 means (plus 2) 3

- if exp1 then exp2 else exp3 is an expression of type T

  \[ \downarrow \]

  \[ \downarrow \]

  \[ \downarrow \]

  Same type T

  Here, one first evaluates \( \text{exp}_1 \) and depending on its result, one obtains \( \text{exp}_2 \) or \( \text{exp}_3 \).

- \( \text{max}i(x, y) = \text{if } x \geq y \text{ then } x \text{ else } y \)

- \( \text{let } \text{decs} \text{ in } \text{exp} \) is used to define a local declaration block \( \text{decs} \), which is only valid in \( \text{exp} \).

\[
\text{roots } a \ b \ c = \text{let } d = \sqrt{a} \text{ in } \begin{cases} 
\frac{-b + d}{e} & \frac{-b - d}{e} \\
\end{cases}
\]

- case \( \text{exp of } \{ \text{pat}_1 \rightarrow \text{exp}_1; \ldots; \text{pat}_n \rightarrow \text{exp}_n \} \)

Here, one tries to match \( \text{pat}_i \) to \( \text{exp} \). If this succeeds, then the result is \( \text{exp}_i \), etc.

\[
\begin{align*}
\text{and True } \gamma & = \gamma \\
\text{and False } \gamma & = \text{False} \\
\text{and } x \ y & = \text{case } x \\
\text{of } \text{True } \rightarrow y \\
\text{False } \rightarrow \text{False}
\end{align*}
\]
\[
\lambda \text{pat}_1 \ldots \text{pat}_n \rightarrow \text{exp}, \ n \geq 1
\]
stands for "\(\lambda\)"

Lambda-expression

stands for the function that takes \(n\) arguments \(\text{pat}_1, \ldots, \text{pat}_n\), and returns the result \(\text{exp}\).

This allows to define "anonymous" functions by just a single expression.

\[
\lambda x \rightarrow 2 \ast x
\]
stands for the function that takes an argument \(x\) and returns \(2 \ast x\)

\[
(\lambda x \rightarrow 2 \ast x) \ 5 = 2 \ast 5 = 10
\]

Type of \(\lambda \text{pat}_1 \ldots \text{pat}_n \rightarrow \text{exp}\): \(\mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \ldots \rightarrow \mathbb{Z}\)

\[
\lambda x \rightarrow 2 \ast x \quad \text{has type } \mathbb{Z} \rightarrow \mathbb{Z}
\]

\[
\lambda (x, y) \rightarrow x + y \quad \text{has type } (\mathbb{Z}, \mathbb{Z}) \rightarrow \mathbb{Z}
\]

Instead of \(\text{plus} \ x \ y = x + y\)
we could define \(\text{plus} = \lambda \ x \ y \rightarrow x + y\)
\( \sim \quad \text{plus} \quad x = \gamma \rightarrow x + \gamma \)