1.1.1 Declarations

1. Introduction to Haskell

1.1. Main Constructs of Haskell

1.2. Prog. techniques in functional languages

1.3. }

1.1. Main Constructs of Haskell

4 Kinds of main constructs: declarations, expressions, patterns, types

1.1.1. Declarations

Haskell prog = sequence of declarations that have to start at the same column

Declarations are either type declarations or function declarations

We define Haskell’s syntax by EBNF-rules where non-terminal symbols are underlined
Comments in Haskell:

{-
  ...
-}

or

until

end of line

Type declarations (Slide 4)

\[ \text{square} :: \text{Int} \rightarrow \text{Int} \]

Variables are used as names of functions

Type of Functions from Int to Int

Haskell has pre-defined types like Int, Bool, ...

One can create new types by "-" or by " [ ... ] :

\[ \text{Int} \rightarrow \text{Bool} \]

\[ [\text{Int}] \]

\[ [\text{Int}] \rightarrow \text{Int} \]

Type declarations do not have to be specified by the programmer (then Haskell computes them automatically). But it is good programming style to provide them.

Function declarations

\[ \text{Square} x = x * x \]

Defining equations specifying a function
Haskell has certain built-in operations:

- Arithmetic operations: `+`, `-`, `*`, `/`, ...
- Comparison operations: `==`, `>`, `<`, `>=`, `<=`, ...
- Boolean operations: `not`, `&&`, `||`, ...

They evaluate to *True* or *False*

**Evaluation of functional programs** (Slide 5)

Program tries to simplify a given expression.

Ex: Square \((12 - 1)\)

Evaluation works by **term rewriting**:

1. Computer searches for a sub-expression such that
   the left-hand side of a defining equation matches
   this sub-expression (i.e., the LHS is equal to
   the sub-expression if one instantiates its variables
   appropriately).
2. Replace the instantiated LHS of the equation
by the instantiated right-hand side.

\[ \text{Square} :: \text{Int} \to \text{Int} \]
\[ \text{Square } x = x \cdot x \]

2 main evaluation strategies:

- **Leftmost innermost (call-by-value)**
  disadvantage: one evaluates all subexpressions, even if they are not needed for the result. This might lead to unnecessary non-termination.

- **Leftmost outermost (call-by-name)**
  disadvantage: duplicated arguments have to be evaluated several times.

Haskell: Lazy Evaluation

Leftmost outermost, but Haskell keeps track of duplicated arguments and only evaluates them once in parallel.

\[ \text{three} :: \text{Int} \to \text{Int} \]
\[ \text{three } x = 3 \]
\[ \text{non-term} :: \text{Int} \to \text{Int} \]
\[ \text{non-term } x = \text{non-term } (x+1) \]

\text{non-term 0} \quad \text{does not terminate}

\text{three (non-term 0)} \quad \text{terminates in 7 evaluation steps, result is 3}

Haskell implementation
We recommend the GHC (Glasgow Haskell Compiler) in particular the GHCi interpreter.

"interactive"

**Conditional equations (Slide 6)**

```
maxi :: (Int, Int) -> Int
maxi (x, y) | x > y = x
             | otherwise = y
```

- RHS of an equation can be restricted by a Boolean condition.
- Conditions are checked from top to bottom, don't have to be exhaustive.
- "otherwise" always evaluates to `True`

**Currying**

Named after logician Haskell B. Curry.

Idea: don't provide several arguments as a tuple, but provide them individually one after another.

```
plus :: Int -> (Int -> Int)
plus x y = x + y
```

Here:

```
(plus 2) 3 = 5
```

(\(\text{Int, Int})\) type of tuples of 2 components, both have type \(\text{Int}\).

\((2, 3)\) has type \((\text{Int, Int})\)
Int \to Int \to Int \text{ stands for } Int \to (Int \to Int) \to \text{associates to the right}

plus \; x \; y \text{ stands for } (\text{plus} \; x) \; y \text{ function application associates to the left}

Advantages of Currying:
- less brackets
- "partial applications" are possible: one may apply a function to fewer arguments

plus \; 2 \text{ is an expression of type } Int \to Int.
It stands for the function that takes an argument and increases it by 2.

\Rightarrow \text{var can now be applied to several arguments pat, \ldots, pat} \text{ (Slide 7)}.

Function definition by pattern matching

\text{\textit{and}} :: \text{Bool} \to \text{Bool} \to \text{Bool}
\text{\textit{and}} \; \text{True} \; y = y
\text{\textit{and}} \; \text{False} \; y = \text{False}

2 \text{ lets declarations for the same \textit{and} symbol ("\text{\textit{and}}")}. Patterns \; (\text{True}, \text{False}, y) \text{ are used to describe the expected arguments.}

\text{Data type} \; \text{Bool} \text{ has 2 data constructors: True, False}
\text{Such functions are used to construct data objects, can't be evaluated further.}
Pattern matching: perform a case analysis depending on the data constructors that the arguments are built with.

To evaluate \( \text{and} \ exp \ exp2 \) one first has to check, whether the LHS of the first equation matches \( \text{and} \ exp \ exp2 \).

If yes: use 1st equation for evaluation
If no: check the next equation
2 equations are checked from top to bottom
One could simplify "and" to:

\[
\begin{align*}
\text{and} :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} & \quad \text{unclear} :: \text{Int} \rightarrow \text{Bool} \\
\text{and} \ True \ y &= y & \text{unclear} \ x &= \text{not} \ (\text{unclear} \ x) \\
\text{and} \ x \ y &= \text{False} & \\
\text{and} \ False \ (\text{unclear} \ 0) &= \text{False} \\
\text{and} \ (\text{unclear} \ 0) \ False &= \text{does not terminate} : \\
\end{align*}
\]

Haskell needs to find out whether the pattern True matches \( \text{unclear} \ 0 \). To this end, it evaluates \( \text{unclear} \ 0 \) step by step and checks whether True matches the result ...

Pattern matching for the data type Bool:

\[
\begin{align*}
\text{Bool} & \rightarrow \text{True} \mid \text{False} \\
\end{align*}
\]

Pattern matching for data type of lists \((\mathcal{L} \circ \mathcal{J})\):
\[
\begin{align*}
\text{len} &: : [\text{a}] \rightarrow \text{Int} \\
\text{len} [\text{} &= 0 \\
\text{len} (x : xs) &= 1 + \text{len} \hspace{1pt} xs
\end{align*}
\]

Evaluation of \( \text{len} \hspace{1pt} [15, 70, 36] \):
\[
\begin{align*}
15 : (70 : (36 : [\text{}))) \quad \text{or} \\
15 : 70 : 36 : [\text{}]
\end{align*}
\]

\( \text{len} [70, 36] = 1 + \text{len} [70, 36] = \ldots = 3 \)

\[
\begin{align*}
\text{Second} &: : [\text{Int}] \rightarrow \text{Int} \\
\text{Second} [\text{} &= 0 \\
\text{Second} (x : [\text{} &= 0 \\
\text{Second} (x : y : xs) &= y
\end{align*}
\]

Defining equations don’t have to be exhaustive. (Slide 7)

Pattern declarations (Slide 8)
can be used to declare constants (functions of arity 0)

\[
\text{plus} \hspace{1pt} x \hspace{1pt} y = x + y
\]
\[ \text{suc } 5 = \text{ plus } 15 = 1 + 5 = 6 \]

A pattern declaration assigns a certain expression to a pattern.

**Local declarations** (Slide 9)

Local declarations are only valid within some other declaration.

Ex: Given \( a, b, c \in \mathbb{Q} \), find \( x \in \mathbb{Q} \) such that

\[
\alpha x^2 + b x + c = 0 \\
\implies -b \pm \sqrt{b^2 - 4ac} \\
x = \frac{2a}{2a} - \frac{b \pm \sqrt{b^2 - 4ac}}{2a} \\
\]

Roots :: Float -> Float -> Float -> (Float, Float)

Roots \( a \ b \ c = ((-b - d)/e, (-b + d)/e) \)

where \( d = \text{sqrt} ((b * b - 4 * a * c)) \)

\( e = 2 * a \) \[ \text{Local declarations, only valid in this rhs} \]

* more readable
* more efficient, because \( d \) and \( e \) are only evaluated once

**Offside-rule for layout in Haskell**

1. The first symbol in a block declares declarations determining the leftmost column of this block.
the leftmost column of this block.

2. A new line starting in this column is a new declaration of the same block.

3. A new line starting further to the right continues the declaration in the line above.

   \[ d = \sqrt{b^2 - 4 \times a \times c} \]

4. A new line starting further to the left means that the current declaration block has ended. The new line does not belong to this declaration block anymore.

\[ \text{roots } a \ b \ c = \ldots \]
\[ \text{where } d = \ldots \]
\[ e = \ldots \]

\[ \text{Square } x = x \times x \]

---

**Operators and Infix Declarations**

Up to now: functions in prefix notation: plus 2 3

Sometimes one wants to use infix notation instead:

2 + 3, 2 \leq 3, 2 = 3, True \lor False, ...

Infix functions are also called operators.
Every infix operator can be turned into a prefix function by using \((\ldots)\): 

\[ (\cdot) \quad 2 \quad [\cdot 3] \quad (\text{is } 2 \cdot [3]) \]

Every prefix fact. can be turned into an infix operator by using `\(\ldots\)`: 

\[ 2 \quad \text{`\(\cdot\)`\ } 3 \quad (\text{is } \text{`\(\cdot\)`\ } 2 \cdot 3) \]

Moreover, one can also define infix operators directly. 2 properties have to be determined for infix operators:

1. **Association** 
   
   \[
   \text{divide} :: \text{Float} \to \text{Float} \to \text{Float} \\
   \text{divide } x \cdot y = x / y
   \]

   \[36 \quad \text{`\(\div\)`\ } 6 \quad \text{`\(\div\)`\ } 2\]

   If `\(\div\)` associates to the left, the result is 

   \[(36 \div 6) \div 2 = 3\]

   If `\(\div\)` associates to the right, the result is 

   \[36 \div (6 \div 2) = 12\]

   Haskell allows **infix-declarations**:

   ```haskell
   default \_ \_ infixl `\(\div\)` \_ \_ infixr `\(\div\)` \_ \_ this means that `\(\div\)` associates to the left resp. to the right
   infix `\(\div\)` \_ \_ this means that `\(\div\)` does not associate, i.e.,
   ```
36 `d` 6 `d` 2 is forbidden

E.g.: \( 	o \) associates to the right \((\text{Int} \to \text{Int} \to \text{Int} \) stands for \(\text{Int} \to (\text{Int} \to \text{Int}) \) \)

function applic. associates to the left
\((\text{plus} 2 \text{3} \) stands for \((\text{plus} 2) \text{3} \) \)

\[\text{Square Square 2 stands for} \]
\[ (\text{Square Square}) \text{2} \] \(\text{\# type error} \)

2. Binding priority
\[ (\%\%): \text{Int} \to \text{Int} \to \text{Int} \quad (\@\@): \text{Int} \to \text{Int} \to \text{Int} \]
\[ x \%\% y = x + y \quad x \@\@ y = x * y \]

What is the result of \( 1 \%\% 2 \@\@ 3 \)

One can specify binding priority by a number between 0, \(\ldots\), 9:

- \(\text{infixl 9} \%\%\)
- \(\text{infixl 8} \@\@\)

Then
\[
(1 \%\% 2) \@\@ 3 = 9 \\
+ * +
\]

prefix functions: strings that start with lower-case symbol (e.g. square)
prefix data constructors: strings that start
with upper-case symbol (e.g. `True`)

infix functions: strings of special symbols not starting with `:`
(e.g., `+`, `=`, `>=`, `@`, `...`)

infix data constructors: strings of special symbols starting
with `:` (e.g., `: `)

Currying is also possible for operators:

6 `divide` has type `Float -> Float`.
It takes an argument `y` and divides 6 by `y`.

`divide` 6 has type `Float -> Float`.
It takes a number `x` and divides `x` by 6.