Haskell 98 in short

- Syntax and type inferencing similar to ML
- Strongly typed
- Allows for pattern matching in definitions
- Uses lazy evaluation
- Definition of infinite lists possible
- Has list comprehensions
- Allows to define sets with a syntax very similar to mathematical set notation

3.1. Representing data

- Functional programming languages cover a wide spectrum of philosophies regarding data types:
  - Early languages followed the \( \lambda \)-calculus and had only a very limited number of basic data types (with overlaps) and lists as way to define complex types
  - Newer languages require type declarations and strong typing, often with a static type system and type inferencing, so that a programmer cannot make any type errors without immediately being told about the error

Basic data types in Haskell

Haskell has the following build-in basic data types:
- \texttt{Bool}: \texttt{False} or \texttt{True}
- \texttt{Char}: the set of Unicode characters
- \texttt{Integer}: arbitrary-precision integer numbers
- \texttt{Int}: fixed-precision integer, at least \([-2^{31}, 2^{31}-1]\)
- \texttt{Float}: real floating-point number, single precision
- \texttt{Double}: \texttt{\ast}, double precision

Building more complex data types

- Haskell allows a programmer to define complex data types using the \texttt{data} statement:
  \begin{verbatim}
  data Mybool = MyFalse | MyTrue
  \end{verbatim}
- We can use the following ways to define complex data types out of other data types (using data constructors):
  - Enumeration
  - Lists
  - Tuples
  - Recursion
  - List comprehension

Enumeration

- Enumeration is similar to the code types of Pascal, i.e. we define directly all elements of a data type by explicitly mentioning them
- Example:
  \begin{verbatim}
  data Weekday = Mo | Tu | We | Th | Fr | Sa | Su
  \end{verbatim}
  Mo, Tu, We, Th, Fr, Sa, Su are the data elements of the type \texttt{Weekday}.
- They are also so-called constructors (they can be used to construct the data type).
- Constructors always begin with a Capital letter!

Lists (I)

- Usage of lists as important data structures is very common in functional programming
- Haskell treats lists somewhere between primitive and complex data types, since it already provides list constructors and does not require to explicitly define lists
- If \( a \) denotes a data type, then \([ a ]\) denotes lists over this data type: \([\texttt{Integer}]\) denotes all lists containing integers
- \( []\) denotes the empty list
- The \texttt{\ast} constructor can also be used to produce lists
Lists (II)
- Examples:
  - `2 : 3 : 5 : 1 : 3` (equivalent to first list)
- Traditionally, `[]` is also called "nil" and `:` "cons"
- Lists of lists and so on are possible:
  - `data MyprimitiveList = [Bool]`
  - `data MycomplexList = [Bool] | [MyprimitiveList]`
- For lists of numbers we can use ranges, which even allows for infinite lists:
  - `data Positives = [0..]`
  - `data Evens = [2, 4..]`

Recursion for defining types
- In Pascal we have seen how records were used to define recursive structures
- In Haskell (and other functional languages) the use of parameters in type declarations and the ability to define constructors allows all kinds of recursively defined types:
  - `data Tree a = Leaf a | InnerNode (Tree a) (Tree a)`
  - The above statement defines binary trees over an arbitrary data type `a`.
- The parameterized definitions will require polymorphism with regard to functions working on `Tree`

List comprehension (I)
- In mathematics it is rather common to define subsets (or subtypes) of types by defining conditions on the elements of a type `a` that the subtype `b` has to fulfill:
  - Even numbers are all numbers that are dividable by 2 without rest
  - A sorted list is a list where each element is smaller or equal to the next element
- This allows instead of writing an algorithm that produces a certain result to just give conditions and let the computer "figure out" how to get there!
  - `f` is declarative programming
- List comprehension introduces this powerful concept into Haskell

List comprehension (II)
- General form (right side of data expression):
  - `[ e | q1,…,qn ]` where `e` is an expression describing the base type that is kind of filtered (or extended) by the `qi` (so-called qualifiers)
- Qualifiers can be:
  - Generators of the form `pattern <- expr` (see later)
  - Guards: arbitrary expressions of type `Bool`
  - Local bindings for variables to be used in later qualifiers

List comprehension (III)
- List comprehensions can be used not only to declare data types but in all kinds of expressions
  - `[n * n | n <- [1..50]]` defines a list (set) of the first 50 cubes
  - `factors n = [ i | i <- [1..n div 2], n mod i == 0 ]` defines a function that finds all factors of the number `n`
  - `data Squares = [n*n | n <- [0..]]` defines the set of all squares
General remarks to data types in Haskell

- The definition of a data type can already include particular computations that in other paradigms would be part of the statements in a procedure or method.
- In order to allow overloading of functions, Haskell has the concept of type classes that help to keep track of what functions have to be defined for a certain type. While there are several similarities to object-oriented classes, the type classes are not describing sets of objects, they are acting more like a Java interface definition. Do not confuse them!

3.2 Control Constructs

- As already seen in the \(\lambda\)-calculus, functional languages usually do not have explicit constructs that provide alternative paths of execution or loops.
- Most functional languages try to follow mathematical notation for defining functions, i.e., they provide several definitions for the function that cover different cases for the input parameters.
- This requires (limited) pattern matching capabilities.
- Some languages nevertheless provide some explicit control constructs to please certain programmers.

Pattern matching in Haskell (I)

- Pattern matching is a generalization of a case distinction, where we do not just check for a particular value, but for a whole structure (defined by the pattern).
- Pattern in Haskell can contain variables (perhaps better termed placeholders) and either predefined or user-defined constructors.
- A given value of a data type fulfills a pattern, if there is a substitution of the variables so that the substituted pattern evaluates to the given value.

Pattern matching in Haskell (II)

- The substitution for the variables is recorded and the variables can be used within the expression containing the pattern as reference to the particular substitutions.
- The pattern \([x, [y, z]], [\text{Prod} 7, (20, 1066)]\) matches \([12, [1, 2]], [\text{Prod} 2 7, (20, 12, 1066)]\) but not \([5, [1, 2]], [\text{Prod} 2 7, (20, 12, 1066)]\) (two different substitutions for \(x\)) and not \([13, [1, 2]], [\text{Prod} 2 7, (20, 13, 1066)]\) (if we assume that the last element is of type Date and we limited Month to 1..12).

Case expressions

- Pattern matching can be directly used to provide several cases of the definition of a function or it can be used within a case expression.
- The general form of a case expression is:
  \[
  \text{case } (x_1, \ldots, x_n) \text{ of } (p_{11}, \ldots, p_{1n}) \rightarrow e_1 \ldots (p_{m1}, \ldots, p_{mn}) \rightarrow e_m
  \]
- Case \(e_i\) of \(\text{True} \rightarrow e_i\) \(\text{False} \rightarrow e_i\)
  defines a if-then-else.

General remarks on control structures in Haskell

- We seem to miss two types of control structures from previous programming paradigms:
  - Combine single manipulations
  - Express repetitions of manipulations
- But functional languages do not do manipulations, they evaluate functions.
- The \(\text{do}\) expressions allow to combine function executions, but they are commonly used only for IO (see 3.7).
- Repetitions are achieved by recursion!