Chapter 1: Introduction

Learning targets of this chapter:

1. Differences between the imperative and functional programming paradigm
2. Benefits of a multitude of programming languages and concepts
3. Referential transparency of the purely functional programming style
4. Historical development of the functional programming paradigm
5. Reusability via functions as parameters
6. Attributes of Haskell: higher-order functions, static typing, referential transparency, non-strict evaluation (laziness)

Imperative and Declarative Languages

1. Imperative Languages
   - Programming = specification of program execution as operations on states (concept of time)
   - Programs specify essentially the control flow
   - Structuring elements: loops, subprograms, abstract data types, ...
   - Object orientation: partitioning of the state, delegation of control to objects, abstraction of computational progress by subprogram calls
2. Declarative Languages
   - Programming = specification of an input/output relation (no concept of time)
   - Control flow remains implicit (not accessible by the programmer); order of computations can be influenced by choice among several options

The Universe of Programming Languages

1. Imperative Languages (Fortran, Algol, Pascal, C, Modula, C++, Java, C#)
2. Declarative Languages
   (a) Logic Languages (Prolog)
   - Automatic search for a computation path via backtracking.

   (b) $\boxed{\text{Functional Languages}}$ (LISP, APL, ML, FP, $\boxed{\text{Haskell}}$, F#)
   - The input/output relation is a function.
   - Programming is function composition.

Why Functional Programming?

- Tackle complex problems by dividing them into subproblems (modularisation).
- Challenge: reuse and combine partial problem solutions.

Answer: Functional Programming

- Functions as data objects and data objects as functions.
- Functional programming permits the adaptation of functions to the problem at hand
  - as the execution proceeds
  - selectively for a specific use
  - type-safely
- In object-oriented programming: inheritance, more flexibly: function objects

Compositionality and Reusability

(Prerequisite: functions as parameters)

- The sort function Mergesort
  - comparison of two elements $<$, $>$, $<_{\text{lex}}$, ...
- A divide-and-conquer scheme for Mergesort of lists
  - function for testing the trivial case: list length $= 1$?
  - function for problem division: list division
  - function for solving the trivial case: identity
  - function for combining the partial solutions: ordered merge

$\boxed{\text{function parameter} \rightarrow \text{functional programming}}$

Programming Thrives on a Multitude of Languages

1. Myth: one programming language (e.g., Java) is sufficient.
   - In principle: every universal (Turing-equivalent) languages suffices.
   - In practice: support for special applications necessary.
2. Myth: language concepts can be identified with programming paradigms.
   - Polymorphism, type classes and inheritance exist also in Haskell.
   - Functional programming does not require the explicit use of recursion (combinators are preferable).
   - Functions as parameters have been present in imperative languages for decades, e.g., in APL (1962) and Pascal (1971).
3. Myth: one has to stick to one paradigm.
   - There are interfaces between programming languages.
   - Blends of paradigms have existed for a long time, e.g., in LISP 1.5 (1965) or Objective Caml (1996). Nowadays: Scala [2003].

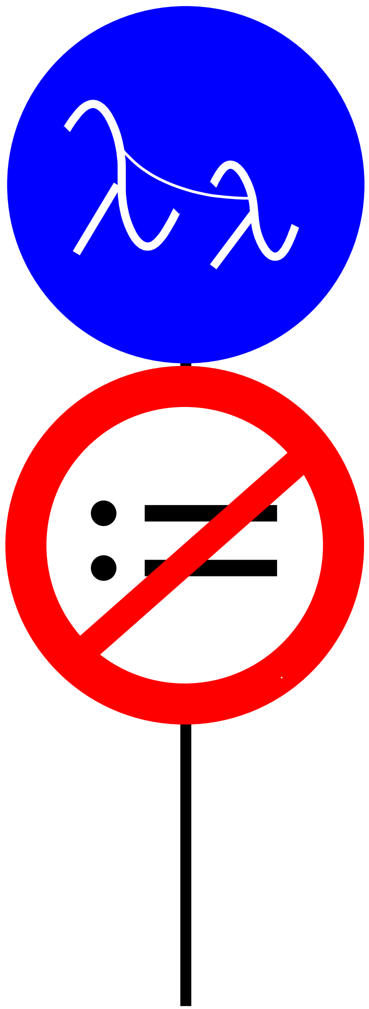
Haskell as Language of Choice in this Course

- Haskell has properties that support safety and productivity
  - non-strict, purely functional language
  - static type system
  - higher-order functions
- Effective, freely available and open-source software tools
- Comfortable syntax, example: pythagorean triples

```
pyth :: Int -> [(Int,Int,Int)]
pyth n = [ (a,b,c) | a<-[1..n], b<-[1..n], c<-[1..n], a^2+b^2==c^2 ]

*Main> pyth 15
[(3,4,5),(4,3,5),(5,12,13),(6,8,10),(8,6,10),(9,12,15),(12,5,13),(12,9,15)]
```

Purely Functional Programming



+ **Compositionality**
+ **Reusability**

− **State changes**
− **Side effects**

Side Effects: Loss of Reusability

Ex.: program in the impurely functional language OCaml

```
open Printf;;
let c = ref 0

let f x = c := !c+1;
          x + !c

let text = let x = 5 in
           c := 0;
           printf "          f(%d) = %d \n"
                               x    (f (x));
           c := 0;
           printf "     f(%d)=f(%d) = %b \n"
                          x     x    (f (x) = f (x));
           c := 0;
           printf "f(%d),f(%d),f(%d) = %d,%d,%d \n"
                     x     x     x    (f (x)) (f (x)) (f (x))
```

Side Effects: Loss of Reusability

Ex.: Program in the impurely functional language OCaml

Program output:

```
val c : int ref = contents = 0
val f : int -> int = <fun>
          f(5) = 6
     f(5)=f(5) = false
f(5),f(5),f(5) = 8,7,6
val text : unit = ()
```

Side Effects: Loss of Reusability

Ex.: Program in the non-functional language Java

```
import java.util.Random;

public class Seiteneffekte {

    static int i=0;

    static int myInt() {
        return i++;
    }

    static int fixedInt() {
        return i;
    }
```

```
static int randomNumber() {
    Random r = new Random(myInt());
    return r.nextInt(10);
}

static void printCharArr(char[]cs){
    int i=0;
    while (cs[i] != '\0') System.out.print(cs[i++]);
    System.out.println();
}

static char[] copy(char[] s) {
    char[] res = new char[s.length];
    int i=0;
    while(s[i]!='\0')    res[i] = s[i++];
    res[i]=s[--i]='\0';
    return res;
}
```

Side Effects: Loss of Reusability

Ex.: Program in the non-functional language Java

```
public static void main(String[] args) {
    int a = myInt();
    int b = myInt();
    System.out.println(a==b);                 // false!

    System.out.println(myInt() == myInt());   // false!

    // we shall not call myInt() in the following code!

    int fix = fixedInt();
    System.out.println("Zufallszahl: " + randomNumber());  // some number
    int fix2 = fixedInt();
    System.out.println(fix==fix2);            // false!
```

```
        char[]cs = {'H','i','\0'};
        printCharArr(cs);                          // Hi
        char[]kopie = copy(cs);
        printCharArr(kopie);                       // Hi
        printCharArr(cs);                          // H

    }

}
```

One Central Benefit of Functional Programming

- Referential Transparency, i.e.,
- Validity of the Leibniz Rule (= Identity Principle):
  two things are equal if and only if they cannot be distinguished.
  Consequence: Substitution Principle: equals can be substituted for each other everywhere.
  Example: if x is defined by the equation x=42, both the name x and the value 42 can be used equally everywhere throughout the program.

Milestones of Functional Programming

1. λ-calculus [Church, Kleene, 1930er]
2. LISP [McCarthy, 1958]
3. APL [IBM, 1962]
4. FP [Backus, 1977]
5. ML [Milner, 1977]
6. Miranda [Turner, 1984]
7. Haskell [Hudak, Wadler u.a., 1990], Standard [1998]
8. MetaOCaml [Sheard, Taha, 2001]
9. FC++ (based on C++ templates) [McNamara, Smaragdakis, 2001]
10. Generic Haskell [Hinze, Jeuring, Löh, 2004]

λ-Calculus

- Computations as *objects* of mathematical operations
  Ex. $(+1) \rightarrow \boxed{(\lambda f \ \rightarrow \ f \circ f)} \rightarrow (+2)$
  implemented in a functional programming language

- Equivalent concepts of computability
  (equational proofs [Church, Kleene, Turing, late 1930s]):
  - λ-Calculus [Church, Kleene, early 1930s]
  - Recursive functions [Gödel, 1934]
  - Turing machines [Turing, 1936]
  - String rewriting systems (Markov, Post, a.o.)

LISP

- Linked lists for symbolic differentiation [McCarthy, 1958]
- LISP 1 (*pure* LISP) [McCarthy, 1960] is functional, many later LISP dialects are not (contain re-assignment)
- Data objects: *S-expressions* (parenthesized atoms)
- Examples (Emacs-LISP):
  - $((\lambda x \to 2(x+1))\,3)$: `((lambda (x) (* 2 (+ x 1))) 3) = 8`
  - `(car (quote ((A B) (C D) (E F)))) = (A B)`
  - `(cdr (quote ((A B) (C D) (E F)))) = ((C D) (E F))`
  - `(append (quote (A B)) (quote (C D))) = (A B C D)`
- CAR/CDR: contents of address/data register (IBM 704)
- `(f x)`: function application, `(quote (f x))`: data object

Backus FP (1)

[Backus, 1977]: Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak:

- their primitive word-at-a-time style of programming inherited from their common ancestor — the von Neumann computer,
- their close coupling of semantics to state transitions,
- their division of programming into a world of expressions and a world of statements,
- their inability to effectively use powerful combination capabilities for building new programs from existing ones, and
- their lack of useful mathematical properties for reasoning about programs.

Backus FP (2)

Ex.: inner product, implementation

1. imperative:

```
c := 0;
for i := 1 step 1 until n do
   c := c + a[i] * b[i];
```

2. functional: $(/+) \circ (\alpha *) \circ \texttt{Trans}$

   - `Trans`: transpose
   - α: apply-to-all
   - $/$: reduce

Backus FP (3)

Ex.: inner product, execution

	$(/+) \circ (\alpha *) \circ \texttt{Trans} : \langle\langle 1,2,3\rangle, \langle 6,5,4\rangle\rangle$
Apply $\circ$	$(/+) : ((\alpha *) : (\texttt{Trans} : \langle\langle 1,2,3\rangle, \langle 6,5,4\rangle\rangle))$
Apply $\texttt{Trans}$	$(/+) : ((\alpha *) : \langle\langle 1,6\rangle, \langle 2,5\rangle, \langle 3,4\rangle\rangle)$
Apply α	$(/+) : \langle * : \langle 1,6\rangle, * : \langle 2,5\rangle, * : \langle 3,4\rangle\rangle$
Apply $*$	$(/+) : \langle 6, 10, 12\rangle$
Apply $/$	$+ : \langle 6, + : \langle 10, 12\rangle\rangle$
Apply $+$	$+ : \langle 6, 22\rangle$
Apply $+$	28

ML

- Metalanguage for the theorem prover *Edinburgh LCF* (Logic for Computable Functions) [Gordon, Milner & Wadsworth 1977]
- First implementation: translated to LISP, then interpreted
- For a long time the language for teaching functional programming
- Features:
  - proof rules as functions, with which one can compute → higher-order functions
  - polymorphic type inference for abstract daya types (Hindley-Milner type system)
  - isolation and handling of exceptions
  - non-functional input/output (no other choice back then)

Haskell

(the central programming language in this course)

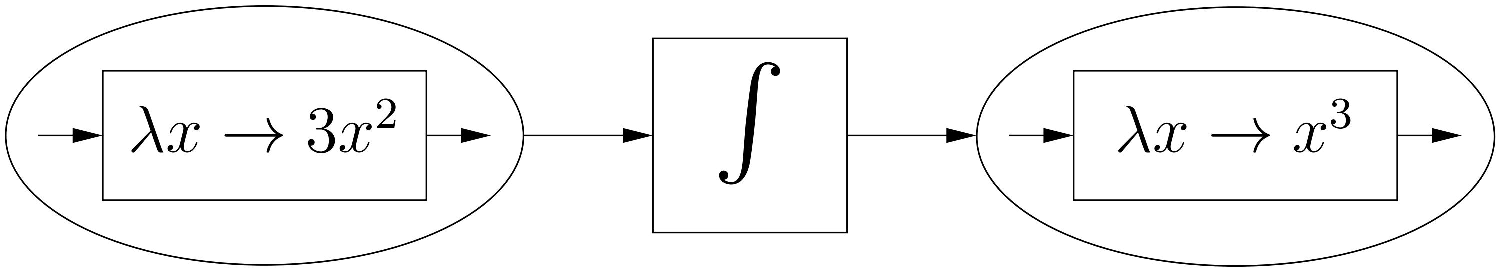
- Freely available language definition, response to the commercial language Miranda
- Design by committee [Hudak, Wadler u.a., 1990]
- Large user community `http://www.haskell.org`
- Many abstraction mechanisms
- Useful software tools, e.g., parser generation, GUIs
- Several implementations available

Properties of Haskell

- Functions as activatable and modifiable data objects
  + high degree of abstraction → reusability
- Purely functional (incl. input/output)
  + simple proofs and transformations of programs
- Static typing
  + comprehensive error prevention *possible*
- Laziness (demand-driven evaluation)
  + avoidance of superfluous computations
  + specification of unbounded and cyclic structures (e.g., hardware)
  − problems of memory consumption if programmed naively

Higher-Order Functions

Ex.: function $\int$ (integral)



Reuse by Generalization

(Ex.: `reduce`)