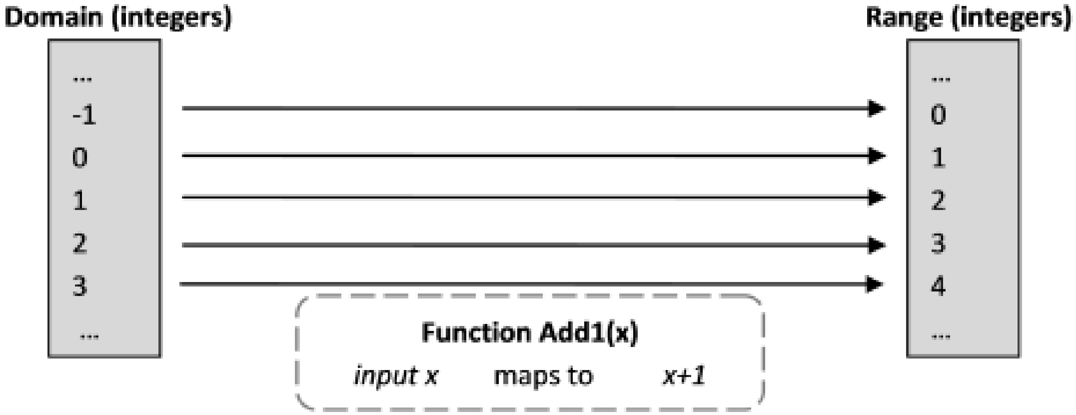
Scientific functional programming

02 – Higher order functions and control flow

Timo Mühlhaus
Computional systems biology

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

A function as instruction



- Domain: The set of values that can be used as input to the function
- Range: The set of possible output values from the function
- **The function is said to map the domain to the range**

Calling a function

- Function declaration
  - Always begin with keyword "let":

```
let sqr y = y * y
val sqr : y:int -> int
```

F#

- Function calling:

```
sqr 5
val it : int = 25
```

F#

Higher-order functions

- *First-order* functions only take and return non-functions

```fsharp
// function signature
int -> int
```
F#

- A *second-order* function can take and return *first-order* functions

```fsharp
// function signature
(int -> int) -> int
```
F#

- *Higher-order* functions can take and return functions of any order
  - They can “glue” functions together to form more complex ones

Higher-order functions as “glue”

- The following infix polymorphic function simply applies it’s second argument to it’s first

```
>> let (|>) x f = f x
val (|>) : ‘a -> (‘a -> ‘b) -> ‘b // signature
```
F#

```
3 |> f //is the same as f (3)
value |> f1 |> f2 |> …
```
F#

- This is called *pipelining*, and is a common style in F#
  - Values flow through the functions in the pipeline from left to right
  - The functions in the pipeline are often formed by partial applications

Currying

- No need to provide all arguments at once

F#

```
let add x y = x + y

let r = add 5 12        // result: 17
let r = (add 5) 12      // result: 17
```

- Partial function application:

F#

```
let add x y = x + y
let addFive = add 5

let r = addFive 12    // result: 17
```

Closures

- Functions can also be constructed at any point in an expression

F#

```
let mapping1D ((aMin, aMax), (bMin, bMax)): float -> float =
     let aRange = aMax – aMin
     let bRange = bMax – bMin
     fun a -> bMin + bRange * (a - aMin) / aRange
```

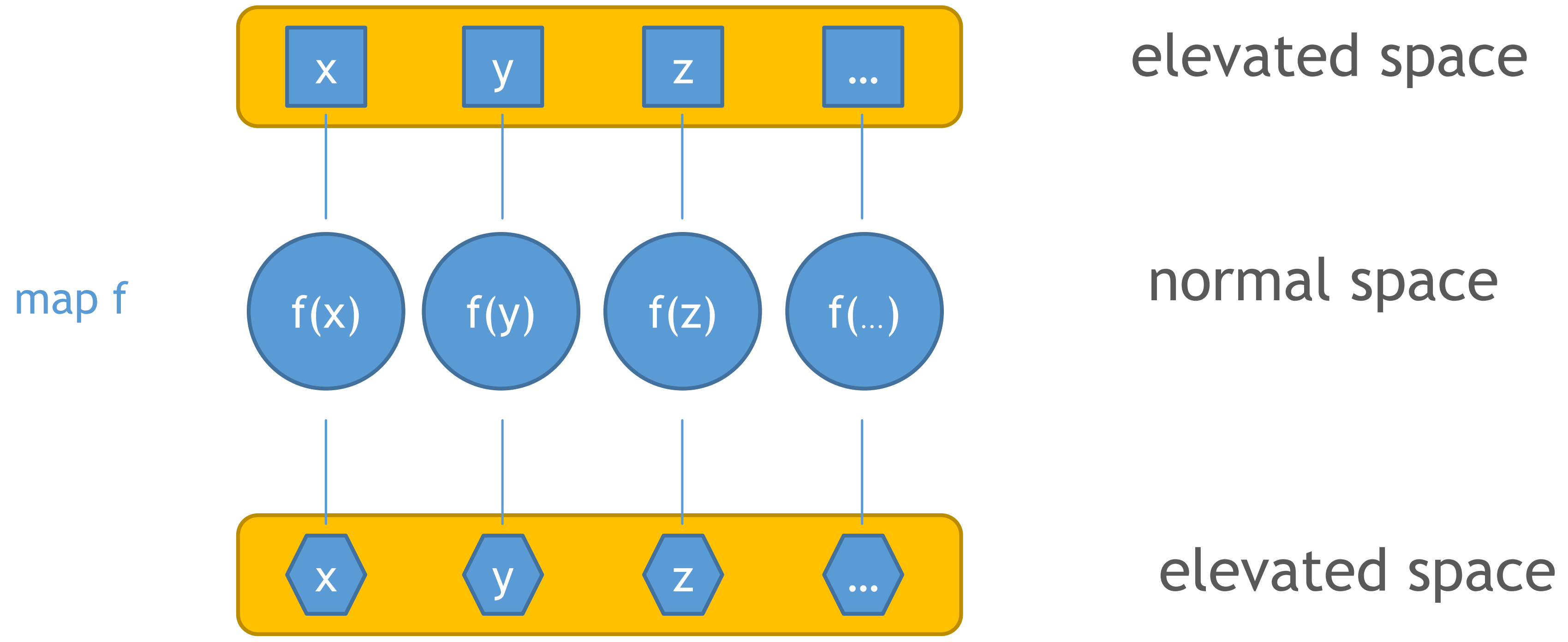
- The function returns a function that can be used many times:

F#

```
let remap = mapping1D ((0.0, 1.0), (3.0, 8.0))
remap 0.5  // return: 5.5
remap 0.8  // return: 7.0
```

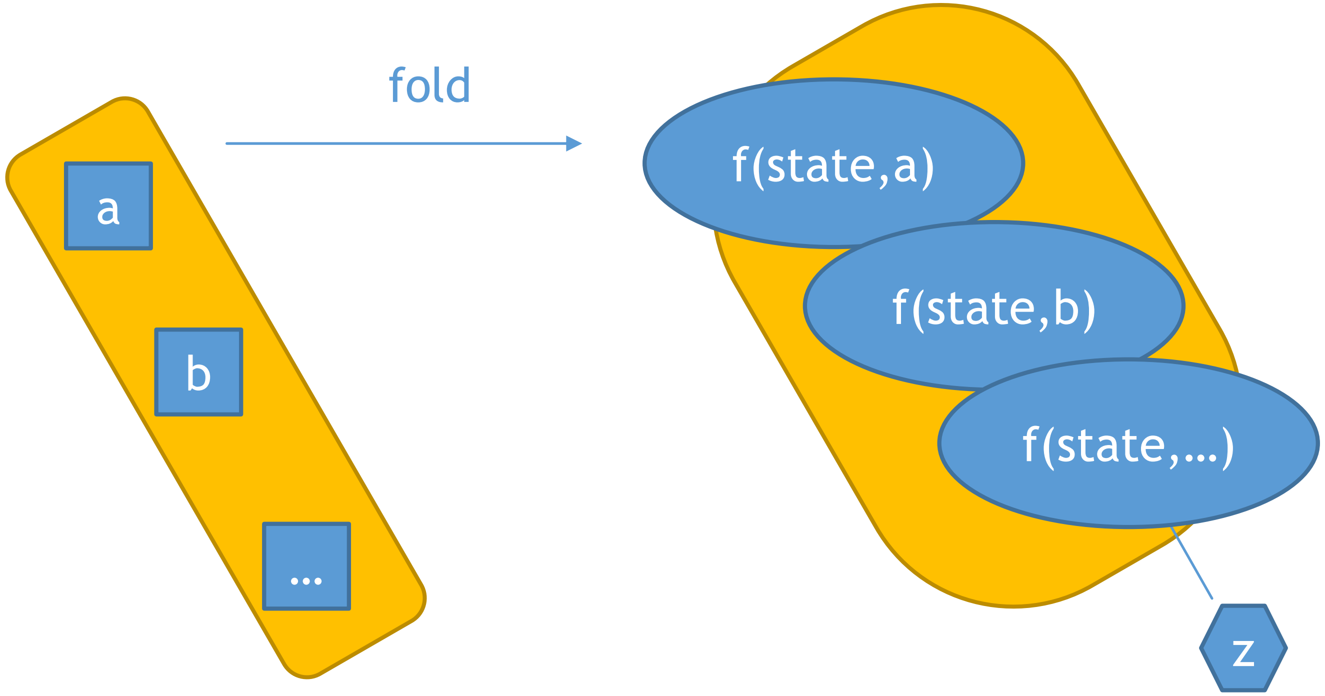
Important conceptual operation: - map -

- The *higher-order* and *polytypic* function ‚map‘ applies a function working on the normal space to an elevated space.

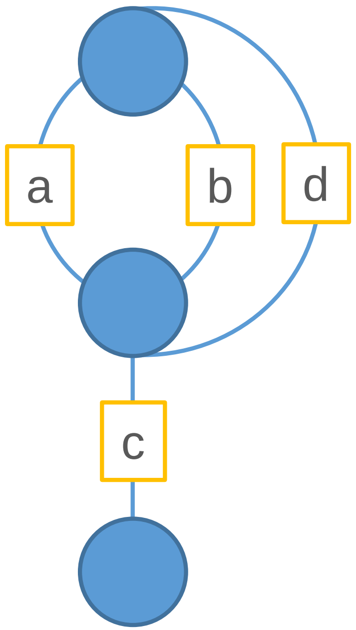


Important conceptual operation: - fold -

- The *higher-order* and *polytypic* function ‚fold‘ applies a function working on the normal space to an elevated space and reduces the elevated space into the normal space (aggregation)



"Control flow" expressions



Flow graph with looping

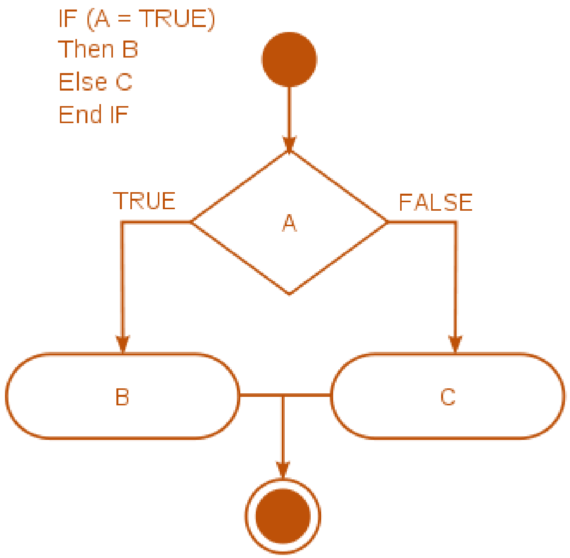
- "Control flow" refers to the decisions made in code that affect the order in which statements are executed.

"Control flow“: if-then-else

- In F# the if-then-else construct is an expression, meaning it returns a value
- Conditional expression can only return boolean values true or false, respectively

```
if condition expression (A) then
    expression B
else
    expression C
```

F#



For-loop expression

- The for-loop iterates either over a loop counter or a sequence of items and executes the body function repeatedly

F#

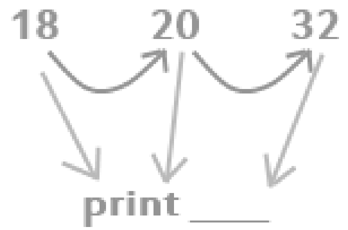
```
let sequence = [18;20;32]
for x in sequence do
   printfn “%i” x
```

…or, using Seq. operation module

F#

```
let sequence = [18;20;32]
sequence
|> Seq.iter printfn “%i”
```

```
seq = [18, 20, 32]
for each x of seq
    print x
end
```

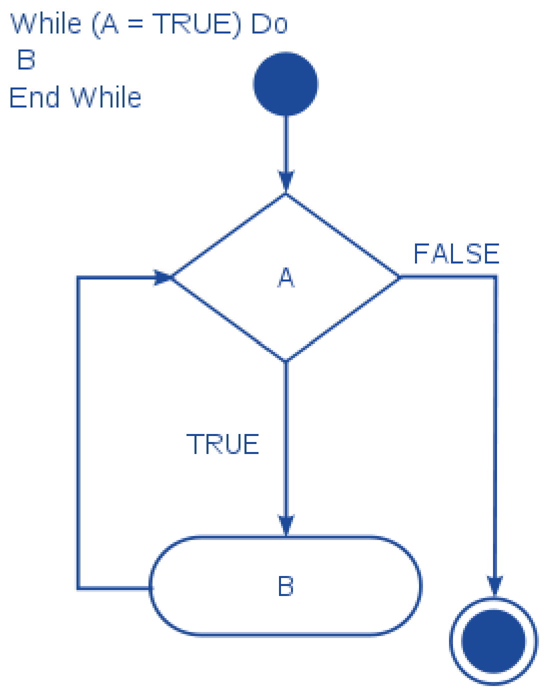


While-loop expression

- The while-loop executes the body function repeatedly while the conditional expression is true.

F#

```
while condition expression (A) do
    expression B
```



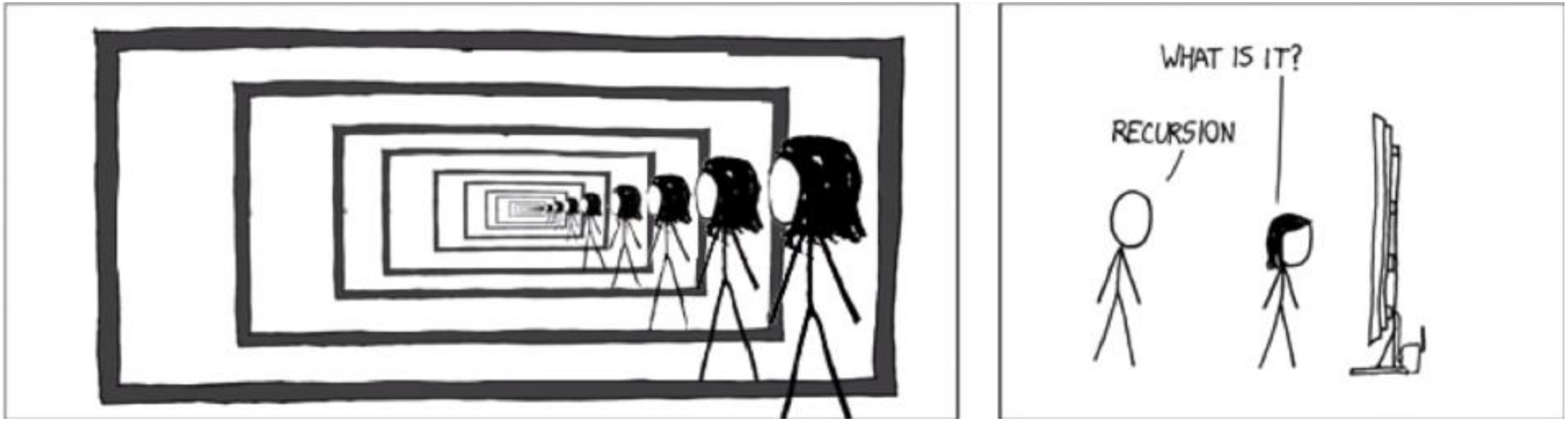
Match expression

- Match expression provides branching control that is based on the comparison of an expression with a set of patterns

F#

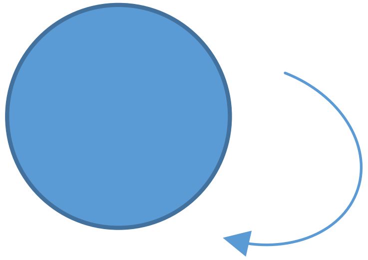
```
let f list =
    match list with
    | [] -> printfn "is empty"
    | x::_ when x > 0 -> printfn "first element is > 0“
    | x::_ -> printfn "first element is <= 0"
```

Recursion



Recursion

- Identity that involves some form of self-reference

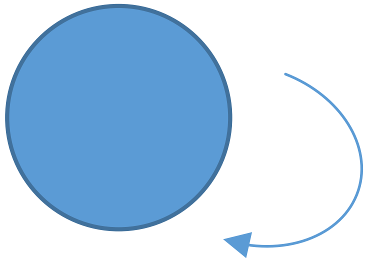


$$x = f(x)$$

"In order to understand recursion, one must first understand recursion,…"

Recursion

- Identity that involves some form of self-reference

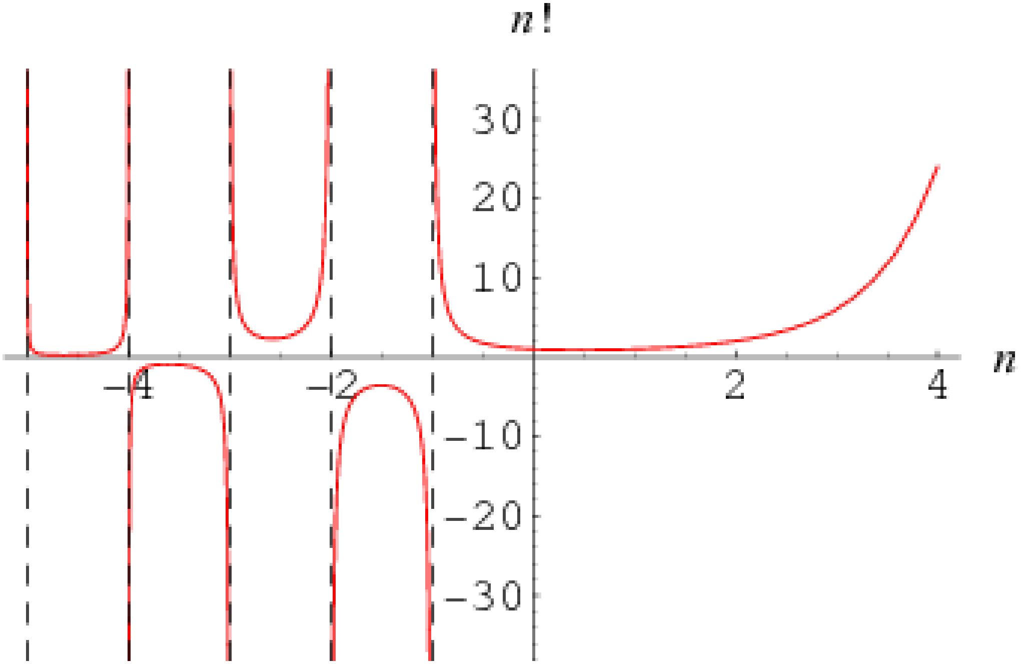


$$x = f(x)$$

"In order to understand *recursion**, one must first understand* *recursion**,…"*

self-reference

Recursion example: factorial n



- In mathematics, the factorial of a non-negative integer n, denoted by n!, is the product of all positive integers less than or equal to n. For example

$$4! = 4 * 3 * 2 * 1 = 24$$

- The value of 0! is 1, according to the convention

Recursion example: factorial n

```
4! = 4 * 3 * 2 * 1 = 24
```

```
4! = 4 * 3!
         3 * 2!
             2 * 1!
                 1
```

Recursion in programming

- Characteristics:
  - Named identity (function)
  - Self reference
  - No mutation
  - Easy to abstract
- Key component
  - Base case
  - Recursive case

base case

recursive case

```
// Function returns the n!
let rec factorial n =
  if n = 1 then
    1
  else
    n * factorial (n-1)
```

self-reference