Functional Programming

Lecture 5 — Recursive definitions

Pedro Vasconcelos
DCC/FCUP

2025

Defining functions

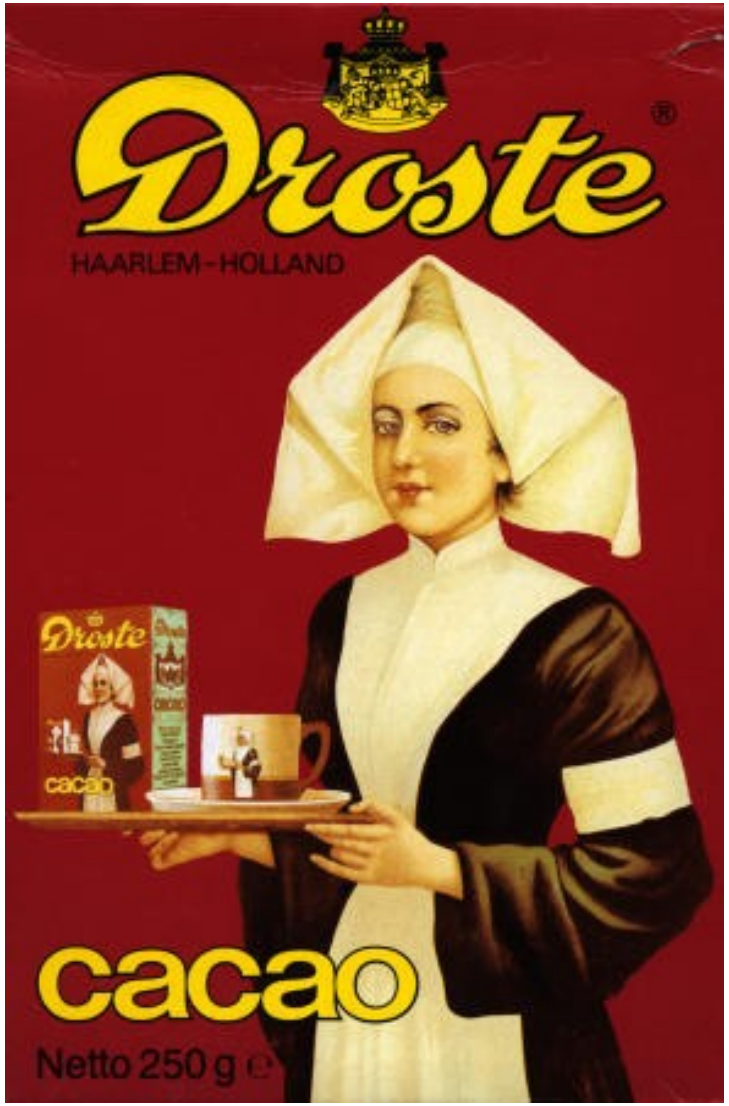
So far we have defined functions only by using pre-existing ones (e.g. from the Prelude).

Example:

```
factorial :: Integer → Integer
factorial n = product [1..n]
```

Defining recursive functions

We can also define functions by *recurrence*, i.e. using the function itself in its definition.



Defining recursive functions (cont.)

Example: factorial defined recursively.

```
factorial :: Integer → Integer
factorial 0 = 1
factorial n = n * factorial (n-1)
```

Step-by-step evaluation

```
  factorial 3
=
  3 * factorial 2
=
  3 * (2 * factorial 1)
=
  3 * (2 * (1 * factorial 0))
=
  3 * (2 * (1 * 1))
=
  6
```

Observations

- The first equation (base case) defines the factorial of 0
- The second equation (recursive case) defines the factorial of n using the factorial of $n - 1$
- If we start from a positive integer, the recursive case *makes progress* towards the base case
- Hence: factorial is defined for all non-negative integers
- If we start from a negative integer, the recursive case *moves away* from the base case

  `factorial (-1)` *non-termination*

Observations (cont.)

- The order of the two equations matters:

```
factorial n = n * factorial (n-1)
factorial 0 = 1
```

  The second equation is never tried, hence this version is *not* defined for any integer!

Syntatical variations

- Two equations, no guards:

```
factorial 0 = 1
factorial n = n * factorial (n-1)
```

- One equation with guards:

```
factorial n | n==0       = 1
            | otherwise = n*factorial (n-1)
```

- One equation with a conditional expression:

```
factorial n = if n==0 then
                  1
              else
                  n*factorial (n-1)
```

Why recursion?

- We can't write loops in a purely functional language because we can't mutate variables!
- Any algorithm that can be writen using imperative loops can be translated into a purely-functional one using recursion
- Moreover, loops can be translated to *tail recursive functions* that optimizing compilers can implement as a loop (more about this later)

Recursion on lists

Because lists are defined recursively, it is quite natural to write list functions using recursion.

Example: the function that computes the product of a list of numbers (from the Prelude).

```
product :: Num a ⇒ [a] → a
product []     = 1
product (x:xs) = x*product xs
```

Example evaluation

```
  product [2,3,4]
=
  2 * product [3,4]
=
  2 * (3 * product [4])
=
  2 * (3 * (4 * product []))
=
  2 * (3 * (4 * 1))
=
  24
```

Non-overlapping patterns

```
product []     = 1
product (x:xs) = x*product xs
```

vs.

```
product (x:xs) = x*product xs
product []     = 1
```

- The two patterns are *non-overlapping*: a list is either empty or non-empty
- Hence the order of the two equations does not matter
- Whenever possible it is preferable to write recursive definitions with non-overlapping patterns

The *length* function

```
length :: [a] → Int
length []     = 0
length (_:xs) = 1 + length xs
```

The *length* function (cont.)

Example evaluation:

```
  length [1,2,3]
=
  1 + length [2,3]
=
  1 + (1 + length [3])
=
  1 + (1 + (1 + length []))
=
  1 + (1 + (1 + 0))
=
  3
```

The *reverse* function

The list reverse function can also be defined recursively; here is a naive definition.

```
reverse :: [a] → [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

(We will see a more efficent version later.)

The *reverse* function (cont.)

Example evaluation:

```
  reverse [1,2,3]
=
  reverse [2,3] ++ [1]
=
  (reverse [3] ++ [2]) ++ [1]
=
  ((reverse [] ++ [3]) ++ [2]) ++ [1]
=
  (([] ++ [3]) ++ [2]) ++ [1]
=
  [3,2,1]
```

Functions with multiple arguments

Functions with multiple arguments can also be defined recursively.

Example: the list concatenation function.

```
(++) :: [a] → [a] → [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

In this case the recursion is only on the *first argument*; the second argument never changes.

Functions with multiple arguments (cont.)

Zip is another function we can define by recursion; in this case both arguments change in each recursive step.

```
zip :: [a] → [b] → [(a,b)]
zip []     _      = []
zip _      []     = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

Functions with multiple arguments (cont.)

Drop is also defined recursively on both arguments.

```
drop :: Int → [a] → [a]
drop 0 xs           = xs
drop n []           = []
drop n (x:xs) | n>0 = drop (n-1) xs
```

Mutual recursion

Sometimes it is useful to define two functions that depend on each other.

Example: testing even and odd natural numbers (in a rather inefficient way).

```
even :: Int → Bool
even 0         = True
even n | n>0 = odd (n-1)

odd :: Int → Bool
odd 0         = False
odd n | n>0 = even (n-1)
```

Quicksort

The *Quicksort* list sorting algorithm can best be specified in a recursive way:

if the input list is empty then it is sorted already

if the input list is non-empty let x be the first values and xs the remaining:

1. recursively sort the values from xs that are less than or equal to x
2. recursively sort the values from xs that are greater than x
3. join the results with x in between

Quicksort (cont.)

In Haskell:

```
qsort :: [Int] → [Int]
qsort []     = []
qsort (x:xs)
    = qsort smaller ++ [x] ++ qsort greater
    where smaller = [y | y←xs, y<=x]
          greater = [y | y←xs, y>x]
```

Probably the most concise implementation of Quicksort in any programming language.

Quicksort (cont.)

Evaluation example (shortening `qsort` as `qs`):

```
  qs [3,2,4,1,5]
=
  qs [2,1] ++ [3] ++ qs [4,5]
=
  (qs [1]++[2]++qs []) ++ [3] ++
  (qs []++[4]++qs [5])
=
  ([1]++[2]++[]) ++ [3] ++ ([]++[4]++[5])
=
  [1,2,3,4,5]
```

Recursion vs. list comprehensions

- We can also translate list comprehensions into recursive functions
- The reverse is not always true: some recursive definitions cannot be expressed as list comprehensions

Recursion vs. list comprehensions (cont.)

Example: list perfect squares from 1 till *n*.

```
-- using list comprehensions
listSquares n = [i*i | i←[1..n]]

-- using recursion
listSquares n = squares 1
   where
     squares i
       | i<=n       = i*i : squares (i+1)
       | otherwise = []
```

Recursion vs. list comprehensions (cont.)

Sometimes we can eliminate the use of lists during the translation.

Example: sum all squares from 1 till *n*.

```
-- using list comprehensions
sumSquares n = sum [i*i | i←[1..n]]

-- using recursion and no lists
sumSquares n = squares 1
   where
     squares i
       | i<=n      = i*i + squares (i+1)
       | otherwise = 0
```

Tail recursion

- A recursive function where recursive calls are always the final expression is called tail recursive
- It is always possible to transform a *for* or *while* loop into a tail recursive function
- An optimizing compiler such as GHC can implement tail recursive functions as loops in machine code

Tail recursion (cont.)

```
# Factorial using a loop
def factorial(n):
  result = 1
  for i in range(2, n+1):
     result = result*i
  return result

-- (Non-tail) recursive factorial
factorialRec 0 = 1
factorialRec n = n * factorialRec (n-1)

-- Tail-recursive factorial
factorialTail n = loop 2 1
  where
    loop i result
          | i<=n      = loop (i+1) (result*i)
          | otherwise = result
```