Programmering og Problemløsning, 2019

Rekursion

Martin Elsman

Department of Computer Science
University of Copenhagen
DIKU

8. oktober, 2019

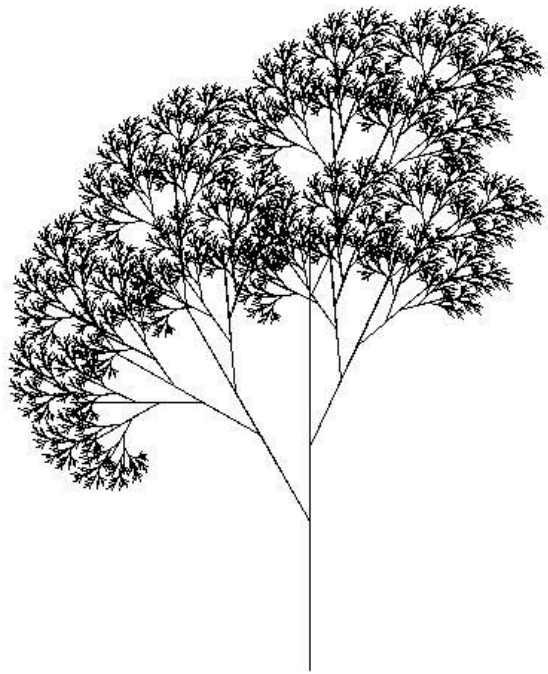
Rekursion

En metode for hvilken en løsning til et problem findes ved at løse mindre instanser af det samme problem.

- Rekursion kan anvendes til at løse en lang række forskellige problemer.
- Rekursion er et af de mest centrale begreber indenfor computer science.

Eksempler på rekursion

- GNU's Not Unix (recursive acronym)
- Google "recursion"
- Mængden af alle heltal:
  $\mathbb{N} = \{n \mid n = 1 \vee \exists k.n = k + 1, k \in \mathbb{N}\}$
- Factorial ($n!$)
- Binær søgning
- Sortering (quick sort, merge sort)
- Find primtal (Eratosthenes si)
- Tegning af fraktaler (tree)



Fakultetsfunktionen ($n!$)

En rekursiv definition:

$$fact(n) = \begin{cases} 1 & n \leq 1 \\ n * fact(n-1) & n > 1 \end{cases}$$

En implementation i F#:

```
let rec fact n = if n <= 1 then 1
                 else n * fact (n-1)
let x = fact 5

// = fact 5
// → 5 * fact (5-1) → 5 * fact 4
// → 5 * (4 * fact (4-1)) ⇝ 5 * (4 * (3 * (2 * 1)))
// → 5 * (4 * (3 * 2)) → 5 * (4 * 6)
// → 5 * 24 → 120
```

Bemærk:

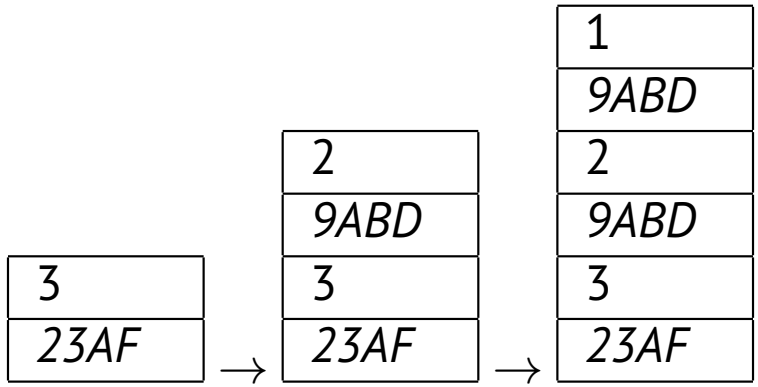
- Nøgleordet **rec** er nødvendigt før en funktion kan henvise til sig selv...

Rekursion i F#

F# oversætteren implementerer rekursion ved at vedligeholde både en *instruktionspeger* (IP) samt en *kald-stak* når programmet kører.

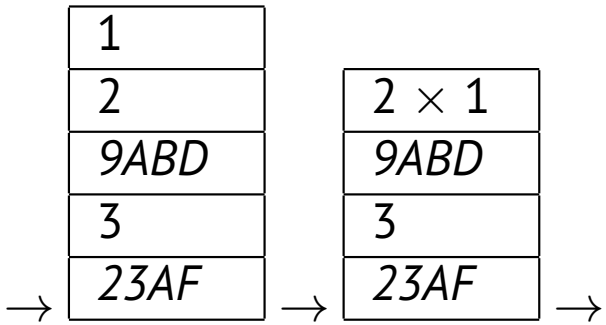
Funktionskald – Ex: `fact 3`:

1. Først lægges returadresse og argument på stakken.
2. IP justeres til adressen på funktionen (og der hoppes).



Funktionsreturnering:

1. Argument tages af stakken.
2. Returadressen tages af stakken.
3. Returværdi lægges på stakken.
4. IP justeres til at indeholde returadressen (og der hoppes).
5. Kalder kan nu læse returværdien på stakken.



Fibonacci-tal

En rekursiv definition:

$$fib(n) = \begin{cases} 1 & n \in \{1, 2\} \\ fib(n-1) + fib(n-2) & n > 2 \end{cases}$$

En implementation i F#:

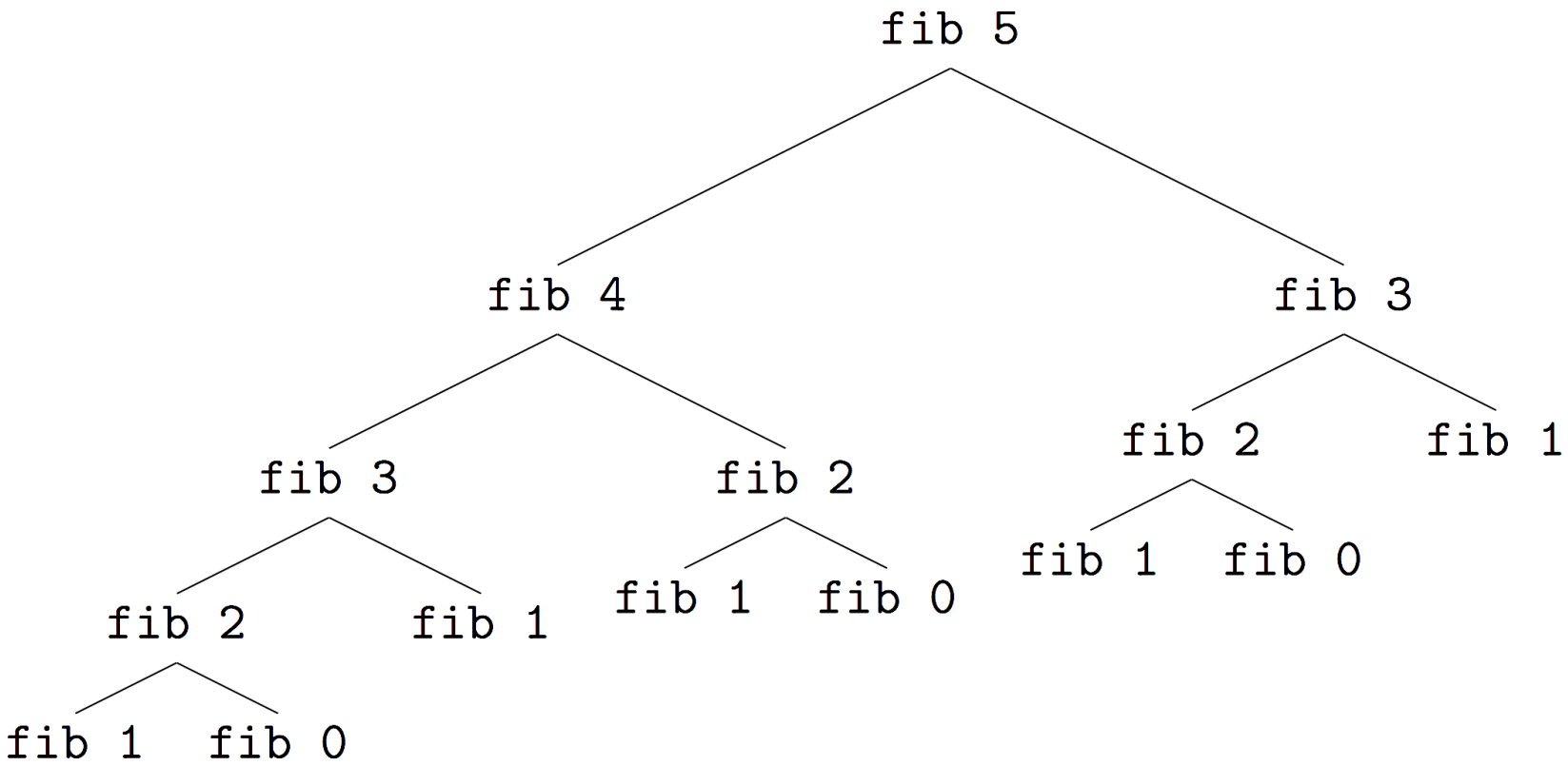
```
let rec fib n = if n = 1 || n = 2 then 1
                else fib (n-1) + fib (n-2)
let x = fib 5

// = fib 5
// → fib(5-1) + fib(5-2) ⇝ fib 4 + fib 3
// → (fib(4-1) + fib(4-2)) + fib 3
// ⇝ (fib 3 + fib 2) + fib 3
// ⇝ ((fib 2 + fib 1) + fib 2) + fib 3 ⇝ ((1 + 1) + 1) + fib 3
// ⇝ 3 + (fib 2 + fib 1) ⇝ 3 + (1 + 1) ⇝ 5
```

Bemærk:

- De 10 første fibonacci tal: `[1; 1; 2; 3; 5; 8; 13; 21; 34; 55]`

Fibonacci-tal (forsat)

Kald-træ:



Spørgsmål:

- Kunne man forestille sig en hurtigere version af `fib`?

Fibonacci-tal (forsat)

En mere effektiv implementation af fibonacci-tal i F#:

```
let rec fib2 p1 p2 n =
  if n = 1 || n = 2 then p2
  else fib2 p2 (p1+p2) (n-1)

let xs = List.init 10 (fun x -> x + 1)
do printf "%A\n" (List.map (fib2 1 1) xs)
```

Kørsel:

```
bash-3.2$ fsharpc --nologo fib.fs && mono fib.exe
[1; 1; 2; 3; 5; 8; 13; 21; 34; 55]
```

Funktionen er "hale-rekursiv"

- Efter det rekursive kald skal funktionen ikke foretage yderligere operationer for at fremkalde resultatet.
- Det betyder at funktionen kører i konstant stak-plads!

Halerekursiv version af fakultetsfunktionen

```
let rec fac2 acc n =
  if n <= 1 then acc
  else fac2 (n*acc) (n-1)

let xs = List.init 10 (fun x -> x + 1)
do printf "%A\n" (List.map (fac2 1) xs)
```

De første ti fakultetstal - [1..10]:

Vi kan også undgå at fakultetsfunktionen bruger unødig stakplads:

```
bash-3.2$ fsharpc --nologo fac2.fs && mono fac2.exe
[1; 2; 6; 24; 120; 720; 5040; 40320; 362880; 3628800]
```

Bemærk:

- Generelt er det en god ide at benytte sig af halerekursion når det er oplagt muligt.

Gensidig Rekursion

F# tillader at man kan definere gensidigt recursive funktioner.

Eksempel:

```
let rec even x =
  if x = 0 then true else odd (x-1)
and odd x =
  if x = 0 then false else even (x-1)

let t = not(odd 34) && even 36 && not(odd 0)
        && even 0 && not(even 1) && odd 1

do printf "%b\n" t
```

Bemærk:

- Nøgleordet **and** benyttes til at knytte de to gensidigt rekursive definitioner sammen.

Rekursion over Lister

Vi kan finde længden på en liste ved hjælp af rekursion:

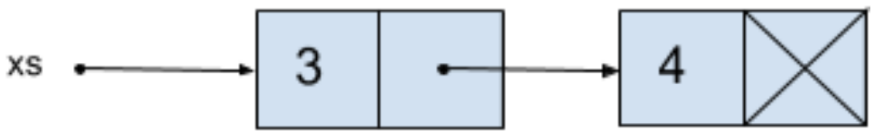
```
let rec length xs =
  if List.isEmpty xs then 0
  else 1 + length (List.tail xs)
```

Bedre hale-rekursiv version (konstant stakplads)

```
let length xs =
  let rec len acc xs =
    if List.isEmpty xs then acc
    else len (acc+1) (List.tail xs)
  in len 0 xs
```

Bemærk:

- Vi vil senere se hvordan "pattern-matching" kan gøre definitionerne endnu mere læselige.



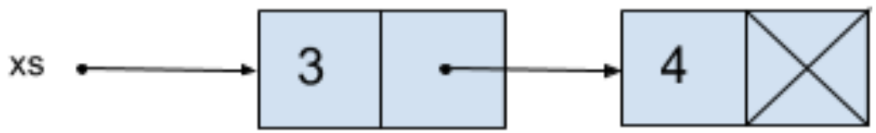
Implementation af `List.find` med rekursion

Vi kan finde et element i en liste direkte ved hjælp af rekursion:

```
let rec find p xs =
  if List.isEmpty xs then None
  else if p (List.head xs) then Some (List.head xs)
       else find p (List.tail xs)

let xs = [34;23;56;76;23]

do printf "%A\n" (find (fun x -> x > 50) xs)
```

De to rekursive tilfælde:

- Når listen er tom — base case (`[]`).
- Når listen har mindst et element (`::`).



Binær søgning i sorteret array

Vi kan finde et element i et **sorteret** array hurtigere end ved at gennemløbe arrayet.

Binær søgning i array:

```
let bsearch (arr:int[]) x =
  let rec bs min max =
       if min > max then None
       else let mid = (max+min) / 2
            in if x < arr.[mid] then bs min (mid-1)
               else if x > arr.[mid] then bs (mid+1) max
               else Some mid
  in bs 0 (Array.length arr - 1)

let arr = [|23;34;41;56;76;123;323|]   // sorteret array
do printf "%A\n" (bsearch arr 76)
```

Kald af funktionen bs:

1. bs 0 7 → mid = 3
2. bs 4 7 → mid = 5
3. bs 4 4 → mid = 4 → Found

⟵ only $log(n)$ steps...