Programmering og Problemløsning, 2019

Typer og Mønstergenkendelse — Part II

Martin Elsman

Datalogisk Institut
Københavns Universitet
DIKU

24. oktober, 2019

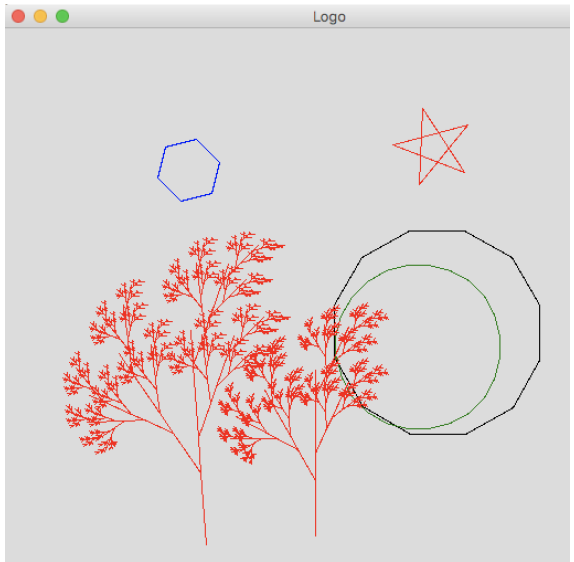
1 Typer og Mønstergenkendelse
- Simple sum-typer
- Sum-typer med argument-bærende konstruktører
- Generiske sum-typer
- Eksempel: Skildpadde EDSL

Sum-Typer

I dag vil vi se på begrebet sum-typer, som er grundlaget for at definere en lang række data-strukturer, såsom træer.

Vi har allerede set eksempler på sum-typer, som inkluderer lister og option-typer.

Sammen med basale datatyper som strenge, heltal og floats og sammen med typer for tupler, records og arrays giver sum-typer os et **komplet væktøjssæt** til at bygge datastrukturer og store applikationer.



1. **Simple sum-typer (enumerations).**
2. **Sum-typer med konstruktører der bærer argumenter.**
3. **Generiske sum-typer.**
4. **Mønstergenkendelse (pattern matching).**
5. **Eksempel: Skildpadde grafik.**

Simple sum-typer

Det er nemt i F# at erklære en sum-type bestående af en mindre mængde af "tokens".

Eksempler:

```
type country = DK | UK | DE | SE | NO
type currency = DKK | EUR | USD | CHF | GBP
type direction = North | South | East | West
```

Pattern-matching:

```
let opposite (dir:direction) : direction =
  match dir with
    | North -> South    // A constructor can appear
    | South -> North    // both as a pattern and as
    | East -> West      // an expression (just as
    | West -> East      // integers can)
```

Bemærk:

- Det kan være en fordel at benytte sig af simple sum-typer i stedet for f.eks. streng- eller heltals-repræsentationer af data.

Sum-typer med argument-bærende konstruktører

Sum-typer kan have konstruktører der tager argumenter.

Eksempel:

```
type object = Pnt | Circ of float | Rect of float * float
```

Pattern-matching er nu mulig:

```
let rec area (obj:object) : float =
  match obj with
  | Pnt -> 0.0
  | Circ r -> System.Math.PI * r * r
  | Rect (a,b) -> a * b
```

Bemærk:

- Det er ikke et krav at alle konstruktører tager argumenter; `Pnt` tager ikke et argument.
- Konstruktører der tager argumenter virker som funktioner i udtryk; f.eks. har `Circ` typen `float -> object`.

Sum-typer kan være type-generiske.

Sum-type definitioner kan være *generiske* således at de er parameteriserede over en eller flere typer.

Denne mulighed kan være brugbar til at skabe genbrugelige konstruktioner.

Option-typen er et godt eksempel:

```
type 'a option = None | Some of 'a
```

Vi kan nu skrive generisk (genbrugelig) kode:

```
let valOf (def:'a) (obj:'a option) : 'a =
  match obj with
  | None -> def
  | Some v -> v
```

Spørgsmål:

- Kender I andre type-generiske sum-typer?

Sum-typer er et meget kraftigt redskab

Sum-typer åbner op for en lang række muligheder:

- Sum-typer kan moduleres med produkter (tupler), men giver mulighed for en mere præcis definition af data-sammenhænge.
- Sum-typer baner vejen for let at definere såkaldte "embeddede" domain-specifikke sprog (EDSLs).
- Sammen med pattern-matching giver sum-typer beriget kontrol over at alle kode-tilfælde er håndteret (jvf. `area` funktionen).

Rekursivt-definerede sum-typer

- Lister kan (som nævnt) forstås som en generisk rekursivt-defineret sum-type med to konstruktører (`[]` og `::`).

```
type 'a list = [] | (::) of 'a * 'a list
```

- Vi skal senere se hvorledes vi også kan definere mere avancerede generiske data-strukturer, såsom træer, ved hjælp af generiske rekursive sum-typer.

Eksempel: Skildpadde EDSL

Som et illustrativt eksempel vil vi se på at definere et skildpaddesprog til brug for at tegne **skildpaddegrafik** (kendt fra programmeringssprog som **Logo**).

Ide:

- Definer et sprog der kan specificere en skildpaddes bevægelser (gå et antal skridt frem, roter et antal grader).
- Giv samtidig mulighed for at skildpadden kan tegne med en pen i en given farve (skildpadden skal kunne skifte pen samt løfte pennen op og ned).

Simpelt domain-specifikt sprog (DSL) som sum-type i F#:

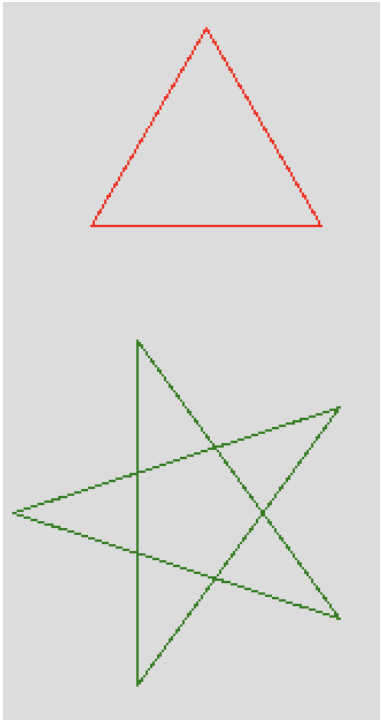
```
type cmd =                  // turtle command
  | SetColor of color         // change pen color
  | Turn of int               // degrees right (0-360)
  | Move of int               // 1 unit = 1 pixel
  | PenUp                     // avoid drawing when moving
  | PenDown                   // draw when moving
```

Eksempler på skildpaddebevægelser

Lad os først antage at vi har en måde hvorpå vi kan "oversætte" en liste af skildpaddebevægelser til et billede.

Skildpaddegrafik:

```
// Draw equal-sided triangle
let triangle x =
  [Turn 30; Move x; Turn 120; Move x;
   Turn 120; Move x; Turn 90]

// Define helper command to repeat
// turtle commands
let rec repeat n cmds =
  if n <= 0 then []
  else cmds @ repeat (n-1) cmds

// Draw a star using 5 lines
let star sz =
  repeat 5 [Move sz; Turn 144]
```



Fortolkning af Skildpaddekommandoer

Før vi kan skrive en "skildpaddekommandofortolker" må vi først bestemme os for hvilke tilstande en skildpadde kan være i.

Skildpaddetilstande:

- Position (type `point = int*int`; initielt i midten af billedet)
- Retning (grader relativt til x-aksen: initielt $^{o}90$).
- Farve (initielt sort på grå baggrund)
- Er pennen oppe eller nede (boolean; `true` hvis oppe)

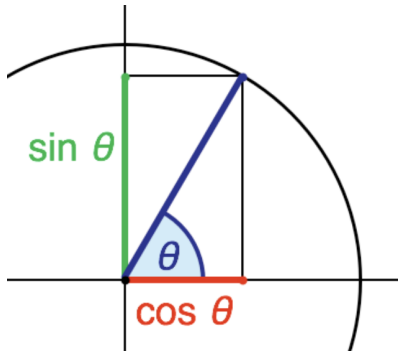
Vi skal også bestemme os for hvad resultatet af "fortolkningen" skal være.

Hvis vi nu kunne fortolke kommandoerne til en **liste af linier** (af type `point*point*color`) vil vi kunne benytte os af `ImgUtil.setLine` til at tegne linierne på et bitmap:

```
val setLine : color -> int*int -> int*int -> bitmap -> unit
```

Skildpaddekommandofortolkningen

```
type point = int * int
type line = point * point * color
let rec interp (p:point,d:int,c:color,up:bool) (cmds:cmd list)
    : line list =
  match cmds with
    | [] -> []
    | SetColor c :: cmds -> interp (p,d,c,up) cmds
    | Turn i :: cmds -> interp (p,d-i,c,up) cmds
    | PenUp :: cmds -> interp (p,d,c,true) cmds
    | PenDown :: cmds -> interp (p,d,c,false) cmds
    | Move i :: cmds ->
      let r = 2.0 * System.Math.PI * float d / 360.0
      let dx = int(float i * cos r)
      let dy = -int(float i * sin r)
      let (x,y) = p
      let p2 = (x+dx,y+dy)
      let lines = interp (p2,d,c,up) cmds
      if up then lines else (p,p2,c)::lines
```



Koden der tegner billedet og viser det i et vindue

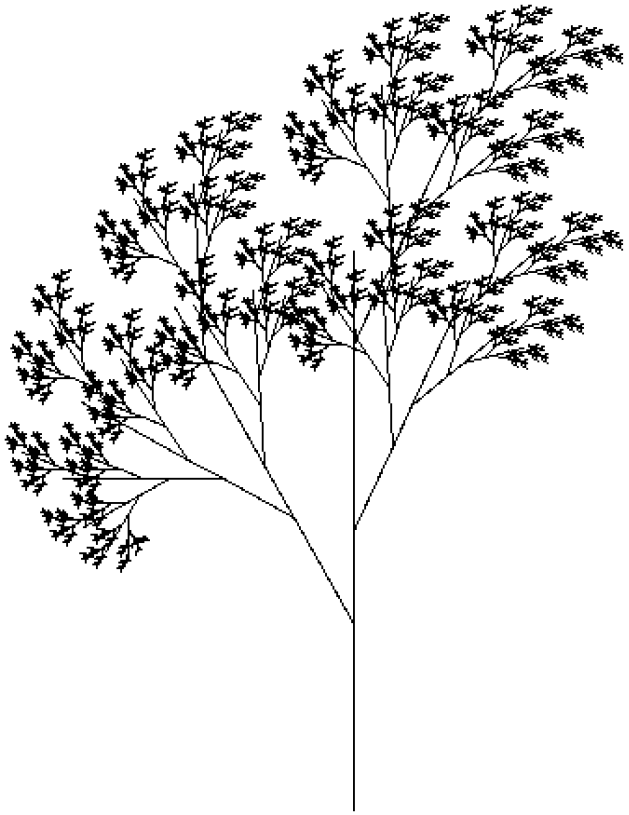
```
let lightgrey = ImgUtil.fromRgb (220,220,220)
let black     = ImgUtil.fromRgb (0,0,0)

let draw (w,h) pic =
  let bmp = ImgUtil.mk w h
  for x in [0..w-1] do                          // initialize
    for y in [0..h-1] do                        // background
      ImgUtil.setPixel lightgrey (x,y) bmp
  let initState = ((w/2,h/2),90,black,false)
  let lines = interp initState pic              // run interp
  for (p1,p2,c) in lines do
    ImgUtil.setLine c p1 p2 bmp                 // update bitmap
  ImgUtil.show "Logo" bmp                       // show bitmap
```

Bemærk:

- Vi benytter os af to nestede for-løkker til at initialisere baggrunden.
- Vi benytter en for-in løkke samt `setLine` til at tegne linierne på bitmap'et.

Træ-fraktalen med skildpaddegrafik

```
let rec tree sz =
  if sz < 5 then
    [Move sz; PenUp; Move (-sz); PenDown]
  else
    [Move (sz/3);
     Turn -30] @ tree (sz*2/3) @ [Turn 30
     Move (sz/6);
     Turn 25] @ tree (sz/2) @ [Turn -25;
     Move (sz/3);
     Turn 25] @ tree (sz/2) @ [Turn -25;
     Move (sz/6); PenUp;
     Move (-sz/3);    // not safe to
     Move (-sz/6);    // reduce these
     Move (-sz/3);    // moves to a
     Move (-sz/6);    // single move!
     PenDown]
```



Spørgsmål: Hvorfor er det ikke en god ide at reducere de sidste fire Move kommandoer til en enkelt Move kommando?