Programmering og Problemløsning

Typer og Mønstergenkendelse (Del 3)

Martin Elsman

Datalogisk Institut, Københavns Universitet (DIKU)

- Simple sum-typer
- Sum-typer med argument-bærende konstruktører
- Generiske sum-typer

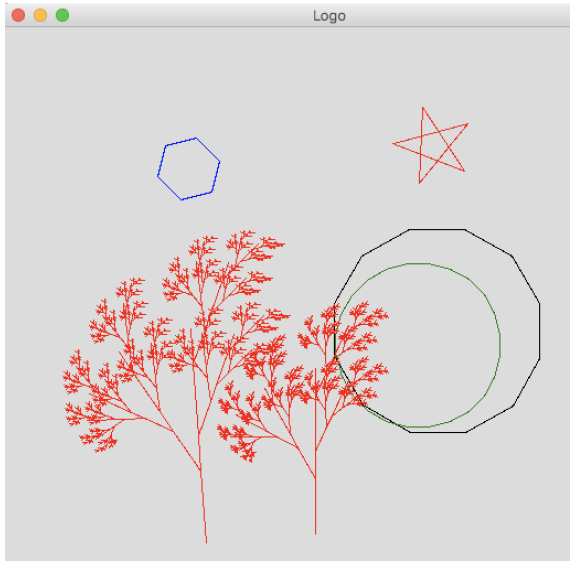
Sum-Typer

Vi skal her 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) for sum-typer.**



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 obj = Pnt | Circ of float | Rect of float * float
```

Pattern-matching er nu mulig:

```
let rec area (obj:obj) : 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 -> obj`.

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

Spørgsmål:

- Nævn en anden type-generisk sum-type?

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.

Konklusion

- Simple sum-typer
- Sum-typer med argument-bærende konstruktører
- Generiske sum-typer