Programmering og Problemløsning
Datalogisk Institut, Københavns Universitet

Jon Sporring

October 18, 2018

1 Lærervejledningn

Emne Træstrukturer og grafik

Sværhedsgrad Middel

2 Introduktion

I de følgende opgaver skal vi arbejde med en træstruktur til at beskrive geometriske figurer med farver. For at gøre det muligt at afprøve jeres opgaver skal I gøre brug af det udleverede bibliotek `img_util.dll`, der blandt andet kan omdanne såkaldte bitmap-arrays til png-filer. Biblioteket er beskrevet i forelæsningerne (i uge 6) og koden for biblioteket ligger sammen med forelæsningsplancherne for uge 6. Her bruger vi funktionerne til at konstruere et bitmap-array samt til at gemme arrayet som en png-fil:

```
// colors
type color = System.Drawing.Color
val fromRgb : int * int * int -> color
// bitmaps
type bitmap = System.Drawing.Bitmap
val mk      : int -> int -> bitmap
val setPixel : color -> int * int -> bitmap -> unit

// save a bitmap as a png file
val toPngFile : string -> bitmap -> unit
```

Funktionen `toPngFile` tager som det første argument navnet på den ønskede png-fil (husk extension). Det andet argument er bitmap-arrayet som ønskes konverteret og gemt. Et bitmap-array kan konstrueres med funktionen `ImgUtil.mk`, der tager som argumenter vidden og højden af billedet i antal pixels, samt funktionen `ImgUtil.setPixel`, der kan bruges til at opdatere bitmap-arrayet før det eksporteres til en png-fil. Funktionen `ImgUtil.setPixel` tager tre argumenter. Det første argument repræsenterer en farve og det andet argument repræsenterer et punkt i bitmap-arrayet (dvs. i billedet). Det tredie argument repræsenterer det bitmap-array, der skal opdateres. En farve kan nu konstrueres

med funktionen `ImgUtil.fromRgb` der tager en triple af tre tal mellem 0 og 255 (begge inklusive), der beskriver hhv. den røde, grønne og blå del af farven.

Koordinaterne starter med $(0,0)$ i øverste venstre hjørne og $(w-1,h-1)$ i nederste højre hjørne, hvis bredde og højde er hhv. w og h. Antag for eksempel at programfilen `testPNG.fsx` indeholder følgende F# kode:

```
let bmp = ImgUtil.mk 256 256
do ImgUtil.setPixel (ImgUtil.fromRgb (255,255,0)) (10,10) bmp
do ImgUtil.toPngFile "test.png" bmp
```

Det er nu muligt at generere en png-fil med navn `test.png` ved at køre følgende kommando:

```
fsharpi -r img_util.dll testPNG.fsx
```

Den genererede billedfil `test.png` vil indeholde et sort billede med et pixel af gul farve i punktet (10,10).

Bemærk, at alle programmer, der bruger `ImgUtil` skal køres eller oversættes med `-r img_util.dll` som en del af kommandoen. Bemærk endvidere, at LaTeX kan inkludere png-filer med kommandoen `includegraphics`.

I det følgende vil vi repræsentere geometriske figurer med følgende datastruktur:

```
type point = int * int // a point (x, y) in the plane
type colour = int * int * int  // (red, green, blue), 0..255

type figure =
  | Circle of point * int * colour
     // defined by center, radius, and colour
  | Rectangle of point * point * colour
     // defined by corners bottom-left, top-right, and colour
  | Mix of figure * figure
     // combine figures with mixed colour at overlap
```

For eksempel kan man lave følgende funktion til at finde farven af en figur i et punkt. Hvis punktet ikke ligger i figuren, returneres `None`, og hvis punktet ligger i figuren, returneres `Some` c, hvor c er farven.

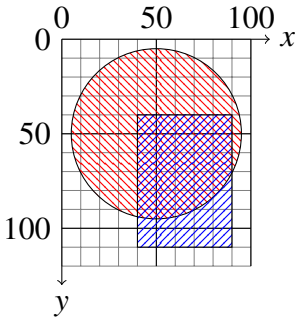
```
// finds colour of figure at point
let rec colourAt (x,y) figure =
  match figure with
  | Circle ((cx,cy), r, col) ->
      if (x-cx)*(x-cx)+(y-cy)*(y-cy) <= r*r
        // uses Pythagoras' formular to determine
        // distance to center
      then Some col else None
  | Rectangle ((x0,y0), (x1,y1), col) ->
     if x0<=x && x <= x1 && y0 <= y && y <= y1
        // within corners
     then Some col else None
  | Mix (f1, f2) ->
      match (colourAt (x,y) f1, colourAt (x,y) f2) with
```

```
    | (None, c) -> c  // no overlap
    | (c, None) -> c  // no overlap
    | (Some (r1,g1,b1), Some (r2,g2,b2)) ->
      // average color
      Some ((r1+r2)/2, (g1+g2)/2, (b1+b2)/2)
```

Bemærk, at punkter på cirklens omkreds og rektanglens kanter er med i figuren. Farver blandes ved at lægge dem sammen og dele med to, altså finde gennemsnitsfarven.

3 Opgave(r)

1. Lav en figur `figTest : figure`, der består af en rød cirkel med centrum i (50,50) og radius 45, samt en blå rektangel med hjørnerne (40,40) og (90,110), som illustreret i tegningen nedenfor (hvor vi dog har brugt skravering i stedet for udfyldende farver.)



2. Brug `ImgUtil`-funktionerne og `colourAt` til at lave en funktion

   ```
   makePicture : string -> figure -> int -> int
                 -> unit
   ```

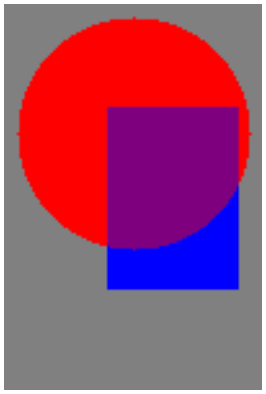
   sådan at kaldet `makePicture` *`filnavn figur b h`* laver en billedfil ved navn *`filnavn`*`.png` med et billede af *`figur`* med bredde *`b`* og højde *`h`*.

   På punkter, der ingen farve har (jvf. `colourAt`), skal farven være grå (som defineres med RGB-værdien (128,128,128)).

   Du kan bruge denne funktion til at afprøve dine opgaver.

3. Lav med `makePicture` en billedfil med navnet `figTest.png` og størrelse 100×150 (bredde 100, højde 150), der viser figuren `figTest` fra Opgave 1.

   Resultatet skulle gerne ligne figuren nedenfor.



4. Lav en funktion `checkFigure : figure -> bool`, der undersøger, om en figur er korrekt: At radiusen i cirkler er ikke-negativ, at øverste venstre hjørne i en rektangel faktisk er ovenover og til venstre for det nederste højre hjørne (bredde og højde kan dog godt være 0), og at farvekompenterne ligger mellem 0 og 255.

   Vink: Lav en hjælpefunktion `checkColour : colour -> bool`.

5. Lav en funktion `boundingBox : figure -> point * point`, der givet en figur finder hjørnerne (top-venstre og bund-højre) for den mindste akserette rektangel, der indeholder hele figuren.

   `boundingBox figTest` skulle gerne give `((5, 5), (95, 110))`.