Pictures

- **Drawing Pictures**
  - Pictures are composed of Regions
  - Regions are composed of shapes
  - Pictures add color and layering

  ```haskell
data Picture = Region Color Region
               | Picture `Over` Picture
               | EmptyPic

  deriving Show
```

- **We need to use SOEGraphics, but SOEGraphics has its own Region datatype.**

  ```haskell
  import SOEGraphics hiding (Region)
  import qualified SOEGraphics as G (Region)
  ```
Recall the **Region** Datatype

```haskell
data Region =
    Shape Shape               -- primitive shape
  | Translate Vector Region   -- translated region
  | Scale      Vector Region   -- scaled region
  | Complement Region         -- inverse of a region
  | Region `Union` Region     -- union of regions
  | Region `Intersect` Region -- intersection of regions
  | Empty
```

- How do we draw things like the intersection of two regions, or the complement of a region? These are hard things to do efficiently. Fortunately, the **G.Region** interface uses lower-level support to do this for us.
G.Region

- The **G.Region** datatype interfaces more directly to the underlying hardware. It is essentially a two-dimensional array or “bitmap”, storing a binary value for each pixel in the window.
Efficient Bit-Map Operations

- There is efficient low-level support for combining bit-maps using a variety of operators. For example, for union:

```
+ =
```

- These operations are fast, but data (space) intensive, and this space needs to be explicitly allocated and de-allocated, a job that seems easier in a much lower-level language.
G.Region Interface

createRectangle :: Point -> Point -> G.Region
createEllipse   :: Point -> Point -> G.Region
createPolygon   :: [Point] -> G.Region

andRegion       :: G.Region -> G.Region -> G.Region
orRegion        :: G.Region -> G.Region -> G.Region
xorRegion       :: G.Region -> G.Region -> G.Region
diffRegion      :: G.Region -> G.Region -> G.Region

drawRegion      :: G.Region -> Graphic

These functions are defined in the SOEGraphics library module.
Drawing G.Region

• To draw things quickly, turn them into a **G.Region**, then turn the **G.Region** into a graphic object, and then use all the machinery we have built up so far to display the object.

```haskell
drawRegionInWindow :: Window -> Color -> Region -> IO ()
drawRegionInWindow w c r =
    drawInWindow w
    (withColor c (drawRegion (regionToGRegion r)))
```

• All we need to define, then, is: **regionToGRegion**.
  – But first, let’s define what it means to draw a picture.
Drawing Pictures

• Pictures combine multiple regions into one big picture. They provide a mechanism for placing one sub-picture on top of another.

\[
\text{drawPic} :: \text{Window} \rightarrow \text{Picture} \rightarrow \text{IO ()}
\]

\[
\text{drawPic } w \ (\text{Region } c \ r) \ = \ \text{drawRegionInWindow } w \ c \ r
\]
\[
\text{drawPic } w \ (\text{p1 `Over` p2}) \ = \ \text{do (drawPic } w \ p2 \ \text{drawPic } w \ p1)
\]
\[
\text{drawPic } w \ \text{EmptyPic} \ = \ \text{return ()}
\]

• Note that \text{p2} is drawn before \text{p1}, since we want \text{p1} to appear “over” \text{p2}.
Summary

• We have a rich calculus of **Shapes**, which we can draw, take the perimeter of, and tell if a point lies within.

• We defined a richer data type **Region**, which allows more complex compositions (intersection, complement, etc.).
  
  – We gave **Region** a mathematical semantics as a set of points in the 2-dimensional plane.
  
  – We defined some interesting operators like $\texttt{containsR}$ which is the characteristic function for a region.
  
  – The rich nature of **Region** makes it hard to draw efficiently, so we use a lower level datatype **G.Region**, which relies on features like overwriting and explicit allocation and deallocation of memory.
  
  – We can think of **Region** as a high-level interface to **G.Region** that hides low-level details.

• We enriched things even further with the **Picture** type, which adds color and layering.
Turning a Region into a G.Region

Experiment with a subset of task to illustrate an efficiency problem. Just consider rectangular shapes and scaling.

\[
\text{regToGReg0} :: \text{Region} \to \text{G.Region}
\]

\[
\text{regToGReg0} \ (\text{Shape} \ (\text{Rectangle} \ sx \ sy))
\]
\[
= \text{createRectangle} \ \text{trans}\ (-sx/2,-sy/2)
\]
\[
\text{trans} \ (sx/2, sy/2)
\]

\[
\text{regToGReg0} \ (\text{Scale} \ (x,y) \ r)
\]
\[
= \text{regToGReg0} \ \text{scaleReg} \ (x,y) \ r
\]

\[
\text{where} \ \text{scaleReg} \ (x,y) \ (\text{Shape} \ (\text{Rectangle} \ sx \ sy))
\]
\[
= \text{Shape} \ (\text{Rectangle} \ (x*sx) \ (y*sy))
\]

\[
\text{scaleReg} \ (x,y) \ (\text{Scale} \ s \ r)
\]
\[
= \text{Scale} \ s \ \text{scaleReg} \ (x,y) \ r
\]
A Problem

• Consider

\[(\text{Scale} \ (x_1,y_1)\)
\(\quad \quad \text{(Scale} \ (x_2,y_2)\)
\(\quad \quad \quad \text{(Scale} \ (x_3,y_3)\)
\(\quad \quad \quad \quad \quad \ldots \ (\text{Shape} \ (\text{Rectangle} \ sx \ sy))\)
\(\quad \quad \quad \quad \ldots \ )\))\]

• If the scaling is $n$ levels deep, how many traversals does \texttt{regToGReg1} perform over the Region tree?
We’ve Seen This Before

• Believe it or not we have encountered this problem before. Recall the definition of reverse:

\[
\begin{align*}
\text{reverse } \ [] & \ = \ [] \\
\text{reverse } (x:xs) & \ = \ (\text{reverse } xs) \ ++ \ [x] \\
\text{where } [] & \ ++ \ zs = zs \\
(y:ys) & \ ++ \ zs = y : (ys ++ zs)
\end{align*}
\]

• How did we solve this? We used an extra accumulating parameter:

\[
\begin{align*}
\text{reverse } xs & \ = \ \text{revhelp } xs \ [] \\
\text{where } \text{revhelp } [] & \ zs = zs \\
\text{revhelp } (x:xs) & \ zs = \ \text{revhelp } xs \ (x:zs)
\end{align*}
\]

• We can do the same thing for Regions.
Accumulate the Scaling Factor

\[
\text{regToGReg1} :: \text{Region} \rightarrow \text{G.Region}
\]

\[
\text{regToGReg1 } r = \text{rToNR} (1,1) \ r
\]

where \[
\text{rToGR} :: (\text{Float,Float}) \rightarrow \text{Region} \rightarrow \text{G.Region}
\]

\[
\text{rToGR} (x_1,y_1) \ (\text{Shape} \ (\text{Rectangle} \ sx \ sy))
\]

= \text{createRectangle}

\[(\text{trans} (-sx*x_1/2,-sy*y_1/2)) \]
\[(\text{trans} (sx*x_1/2, sy*y_1/2)) \]

\[
\text{rToGR} (x_1,y_1) \ (\text{Scale} \ (x_2,y_2) \ r)
\]

= \text{rToGR} (x_1*x_2,y_1*y_2) \ r

- To solve our original problem, repeat this for all the constructors of \textbf{Region} (not just \textbf{Shape} and \textbf{Scale}). We also need to handle translation as well as scaling.
Final Version

```haskell
regToGReg2 :: Vector -> Vector -> Region -> G.Region
regToGReg2 loc sca (Shape s) = shapeToGRegion loc sca s
regToGReg2 (x,y) sca (Translate (u,v) r)
  = regToGReg2 (x+u, y+v) sca r
regToGReg2 loc (x,y) (Scale (u,v) r)
  = regToGReg2 loc (x*u, y*v) r
regToGReg2 loc sca Empty = createRectangle (0,0) (0,0)
regToGReg2 loc sca (r1 `Union` r2)
  = let gr1 = regToGReg2 loc sca r1
      gr2 = regToGReg2 loc sca r2
          in orRegion gr1 gr2
```

- Assuming, of course, that we can define:
  
  `shapeToGRegion :: Vector -> Vector -> Shape -> G.Region`

  and write rules for `Intersect`, `Complement` etc.
A Matter of Style

• While the function on the previous page shows how to solve the problem, there are several stylistic issues that could make it more readable and understandable.

• The style of defining a function by patterns becomes cluttered when there are many parameters (other than the one which has the patterns).

• The pattern of explicitly allocating and deallocating (bit-map) G.Region’s will be repeated in cases for intersection and for complement, so we should abstract it, and give it a name.
Abstract the Low-Level Bit-Map Details

\[
\text{primGReg loc sca r1 r2 op} = \text{let } \text{gr1} = \text{regToGReg loc sca r1} \\
\text{gr2} = \text{regToGReg loc sca r2} \\
in \text{op gr1 gr2}
\]
Redo with a Case Expression

\[
\text{regToGReg} :: \text{Vector} \to \text{Vector} \to \text{Region} \to \text{G.Region}
\]

\[
\text{regToGReg} \ (\text{loc}@(x,y)) \ (\text{sca}@((a,b))) \ \text{shape} =
\]

\[
\begin{cases}
\text{Shape } s & \Rightarrow \text{shapeToGRegion loc sca s} \\
\text{Translate } (u,v) \ r & \Rightarrow \text{regToGReg} \ (x+u, \ y+v) \ \text{sca} \ r \\
\text{Scale } (u,v) \ r & \Rightarrow \text{regToGReg} \ \text{loc} \ (a*u, \ b*v) \ r \\
\text{Empty} & \Rightarrow \text{createRectangle} \ (0,0) \ (0,0) \\
\text{r1 `Union` r2} & \Rightarrow \text{primGReg} \ \text{loc} \ \text{sca} \ r1 \ r2 \ \text{orRegion} \\
\text{r1 `Intersect` r2} & \Rightarrow \text{primGReg} \ \text{loc} \ \text{sca} \ r1 \ r2 \ \text{andRegion} \\
\text{Complement } r & \Rightarrow \text{primGReg} \ \text{loc} \ \text{sca} \ \text{winRect} \ r \ \text{diffRegion}
\end{cases}
\]

where \(\text{winRect :: Region}\)

\[
\text{winRect} = \text{Shape} \ (\text{Rectangle} \\
\quad (\text{pixelToInch xWin}) \ (\text{pixelToInch yWin}))
\]

\[
\text{regionToGRegion} :: \text{Region} \to \text{G.Region}
\]

\[
\text{regionToGRegion} \ r = \text{regToGReg} \ (0,0) \ (1,1) \ r
\]
Shape to G.Region: Rectangle

```haskell
shapeToGRegion1 :: Vector -> Vector -> Shape -> G.Region
shapeToGRegion1 (lx,ly) (sx,sy) (Rectangle s1 s2)  
  = createRectangle (trans(-s1/2,-s2/2)) (trans (s1/2,s2/2))  
  where trans (x,y) = ( xWin2 + inchToPixel ((x+lx)*sx),  
                      yWin2 - inchToPixel ((y+ly)*sy) )
```

[(xWin, yWin), (xWin2, yWin2)]
shapeToGRegion1 \((lx, ly) (sx, sy) (Ellipse \ r1 \ r2)\)

\[= \text{createEllipse}\ (\text{trans}\ (-r1, -r2))\ (\text{trans}\ (\ r1, \ r2))\]

where \(\text{trans}\ (x, y) = \)

\[
(x_{\text{Win2}} + \text{inchToPixel}\ ((x+lx)*sx), \ y_{\text{Win2}} - \text{inchToPixel}\ ((y+ly)*sy))
\]
Polygon and RtTriangle

\[\text{shapeToGRegion1} (lx,ly) (sx,sy) (\text{Polygon pts}) = \text{createPolygon} (\text{map trans pts})\]
where trans \((x,y) = \)
\[
(xWin2 + \text{inchToPixel} ((x+lx)*sx),
yWin2 - \text{inchToPixel} ((y+ly)*sy))
\]

\[\text{shapeToGRegion1} (lx,ly) (sx,sy) (\text{RtTriangle s1 s2}) = \text{createPolygon} (\text{map trans} [(0,0),(s1,0),(0,s2)])\]
where trans \((x,y) = \)
\[
(xWin2 + \text{inchToPixel} ((x+lx)*sx),
yWin2 - \text{inchToPixel} ((y+ly)*sy))
\]
A Matter of Style, 2

- `shapeToGRegion1` has the same problems as `regToGReg1`
  - The extra parameters obscure the pattern matching.
  - There is a repeated pattern: we should give it a name.

```haskell
shapeToGRegion (lx,ly) (sx, sy) s = case s of
  Rectangle s1 s2  -> createRectangle (trans (-s1/2,-s2/2))
                     (trans ( s1/2, s2/2))
  Ellipse r1 r2    -> createEllipse   (trans (-r1,-r2))
                     (trans ( r1, r2))
  Polygon pts      -> createPolygon (map trans pts)
  RtTriangle s1 s2 -> createPolygon
                     (map trans [(0,0),(s1,0),(0,s2)])
where trans (x,y) = ( xWin2 + inchToPixel ((x+lx)*sx),
                   yWin2 - inchToPixel ((y+ly)*sy) )
```
Drawing Pictures, Sample Regions

draw :: Picture -> IO ()
draw p = runGraphics (  
    do w <- openWindow "Region Test" (xWin,yWin)  
       drawPic w p  
       spaceClose w  
    )

r1 = Shape (Rectangle 3 2)
r2 = Shape (Ellipse 1 1.5)
r3 = Shape (RtTriangle 3 2)
r4 = Shape (Polygon [(-2.5,2.5), (-3.0,0),
                      (-1.7,-1.0),
                      (-1.1,0.2), (-1.5,2.0)] )
Sample Pictures

\[
\begin{align*}
\text{reg1} &= r3 \ `\text{Union}` \\
&\quad r1 \ `\text{Intersect}` \\
&\quad \text{Complement } r2 \ `\text{Union}` \\
&\quad r4 \\
\text{pic1} &= \text{Region Cyan } \text{reg1} \\
\text{Main1} &= \text{draw } \text{pic1}
\end{align*}
\]

Recall the precedence of \text{Union} and \text{Intersect}
More Pictures

\[
\text{reg2} = \text{let circle} = \text{Shape (Ellipse 0.5 0.5)} \\
\text{square} = \text{Shape (Rectangle 1 1)} \\
\text{in (Scale (2,2) circle)} \\
\quad \text{`Union` (Translate (2,1) square)} \\
\quad \text{`Union` (Translate (-2,0) square)} \\
\text{pic2 = Region Yellow reg2} \\
\text{main2 = draw pic2}
\]
Another Picture

pic3 = pic2 `Over` pic1
main3 = draw pic3
Separate Computation From Action

oneCircle = Shape (Ellipse 1 1)
manyCircles
    = [ Translate (x,0) oneCircle | x <- [0,2..] ]
fiveCircles =
    foldr Union Empty
        (take 5 manyCircles)
pic4 = Region Magenta
    (Scale (0.25,0.25)
        fiveCircles)
main4 = draw pic4
Ordering Pictures

\[
pictToList :: \text{Picture} \rightarrow [(\text{Color}, \text{Region})]
\]

\[
pictToList \ \text{EmptyPic} \ = \ [] \\
pictToList \ (\text{Region} \ c \ r) \ = \ [(c,r)] \\
pictToList \ (p1 \ `\text{Over}` \ p2) \\
\quad \quad = \ pictToList \ p1 \ ++ \ pictToList \ p2
\]

\[
pic6 = \text{pic4 `Over` pic2 `Over` pic1 `Over` pic5} \\
pictToList \ pic6 \quad \quad \rightarrow \quad \quad [(\text{Magenta,?}), \ (\text{Yellow,?}), (\text{Cyan,?}), (\text{Cyan,?})]
\]

Recovers the Regions from top to bottom.
Possible because Picture is a datatype that can be analyzed.
Two ways of drawing a picture

\begin{align*}
pictToList\;\text{EmptyPic} &= [] \\
pictToList\;(\text{Region } c\; r) &= [(c, r)] \\
pictToList\;(p1 \ `\text{Over}`\; p2) &= \text{pictToList p1} \ ++ \ \text{pictToList p2} \\
drawPic\;w\;(\text{Region } c\; r) &= \text{drawRegionInWindow w } c \; r \\
drawPic\;w\;(p1 \ `\text{Over}`\; p2) &= \text{do } \{ \text{drawPic w p2} \\
&\quad ; \text{drawPic w p1}\} \\
drawPic\;w\;\text{EmptyPic} &= \text{return ()}
\end{align*}

• Something interesting to prove:
\begin{align*}
\text{drawPic w} &= \text{sequence} \cdot \\
&\quad (\text{map (uncurry (drawRegionInWindow w)))} \cdot \\
&\quad \text{reverse} \cdot \\
&\quad \text{pictToList}
\end{align*}
Pictures that React

• Find the topmost Region in a Picture that “covers” the position of the mouse when a left button click appears.
• Search the picture list for the first Region that contains the mouse position.
• Re-arrange the list, bringing that one to the top.

adjust :: [(Color,Region)] -> Vertex -> (Maybe (Color,Region), [(Color,Region)])
adjust [] p = (Nothing, [])
adjust ((c,r):regs) p =
  if r `containsR` p
  then (Just (c,r), regs)
  else let (hit, rs) = adjust regs p
       in (hit, (c,r) : rs)
Doing it Non-recursively

\[
\text{adjust2 \ regs \ p} = \text{case (break (\(_\),r) \rightarrow r \ `\text{containsR}` \ p) \ regs)} \text{ of}
\]

\[
\quad (\text{top,\text{hit}\,\\text{rest}}) \rightarrow (\text{Just \ hit, \ top++rest})
\]
\[
\quad (_{,[]}]) \rightarrow (\text{Nothing, \ []})
\]

This is from the Prelude:

\text{break} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow ([a],[a])

For example:

\text{break even} \ [1,3,5,4,7,6,12] = ([1,3,5],[4,7,6,12])
Putting it all Together

loop :: Window -> [(Color,Region)] -> IO ()
loop w regs =
  do clearWindow w
    sequence [ drawRegionInWindow w c r |
      (c,r) <- reverse regs ]
    (x,y) <- getLBP w
    case (adjust regs (pixelToInch (x - xWin2),
                   pixelToInch (yWin2 - y))) of
      (Nothing, _) -> closeWindow w
      (Just hit, newRegs) -> loop w (hit : newRegs)

draw2 :: Picture -> IO ()
draw2 pic = runGraphics (do w <- openWindow "Picture demo" (xWin,yWin)
                             loop w (pictToList pic))
Try it Out

\[
p1, p2, p3, p4 :: \text{Picture} \\
p1 = \text{Region Magenta } r1 \\
p2 = \text{Region Cyan } r2 \\
p3 = \text{Region Green } r3 \\
p4 = \text{Region Yellow } r4 \\
\]

\[
pic :: \text{Picture} \\
pic = \text{foldl Over EmptyPic } [p1, p2, p3, p4] \\
main = \text{draw2 pic} \\
\]
A Matter of Style, 3

\[
\text{loop2 } w \text{ regs} \\
\quad = \text{ do } \text{clearWindow } w \\
\quad \quad \text{sequence } [ \drawRegionInWindow w \ c \ r \ |  \\
\quad \quad \quad \quad \quad (c,r) \gets \text{reverse } \text{regs} ] \\
\quad \quad (x,y) \gets \text{getLBP } w \\
\quad \quad \text{let } \text{aux } (_,r) = r \ `\text{containsR}` \\
\quad \quad \quad \quad \quad (\text{pixelToInch } (x-xWin2), \\
\quad \quad \quad \quad \quad \text{pixelToInch } (yWin2-y)) \\
\quad \quad \text{case } \text{(break aux } \text{regs)} \text{ of} \\
\quad \quad \quad (_,[]) \quad \rightarrow \text{closeWindow } w \\
\quad \quad \quad \text{(top,}\text{hit:bot}) \rightarrow \text{loop } w \ (hit : (\text{top++bot})) \\
\quad \text{draw3 } \text{pic} = \text{runGraphics (} \\
\quad \quad \quad \text{do } w \gets \text{openWindow } "\text{Picture demo}" \ (xWin,yWin) \\
\quad \quad \quad \text{loop2 } w \ (\text{pictToList } \text{pic}) \\
\quad \)}