Domain Specific Languages

- Use of libraries in programming is ubiquitous - can capture the styles of problem solving in some domain
- Like mini programming languages
- Then there are DSLs
- **Refers to a small not-general purpose language**
    - Captures some specific problem domain e.g. **unix shell, SQL, TEX etc.**
        - Someone who knows the domain should already know the semantics
    - They make programs that are
        - Concise, easy to write and maintain
        - Easy to reason about
        - Something that non-programmers can maintain
    - **But**
        - Language design is hard, people want lots of features, good performance
        - We will end up with lots of lexers, parsers, type checkers etc
- **If we embed the DSL in haskell…**
    - Powerful, easy to maintain domain-specific expression
    - Full haskell expresiveness outside the domain
- **Types of language embedding**
    - **Shallow**
        - Represent DSL programs as values in the host language (e.g. functions)
        - Provide *fixed* semantics
        - A program in the DSL might consist of calls to library functions
    - **Deep**
        - Represent the DSL programs as values in the host language **but kept abstract**
        - **Use higher order functions (combinators) to piece together programs**
        - **A program in the DSL may consist of construction of a value which describes the program that is fed to an interpreter**
- **Building a DSL for images**
    - Most basic element is to draw a shape
    -
      ```
      data Shape = ...
      empty, circle, square :: Shape
      ```
    - To reason about the shapes position or size we need a way to represent coordinates and vectors
    -
      ```
      data Vector = Vector Double Double
      type Point  = Vector
      ```

- And now we need a way to make use of this information to move and deform the shapes

```
data Transform = ...
identity :: Transform
translate :: Vector -> Transform
scale :: Vector -> Transform
rotate :: angle -> Transform
compose :: Transform -> Transform

(<+>) = compose
```

- So now we can draw something

```
type Drawing = [(Transform,Shape)]

example = [ (scale (point 0.5 0.5) <+> translate (point 1.2 0.4), circle) ]
```

- What might an interpretation function look like for a drawing? Perhaps we would ask each point if it is within our drawing...

```
inside :: Point -> Drawing -> Bool
```

- **But this is all just the API**
- **What about the actual implementation?**
    - **Shallow Embedding**
        - Often easier if you can get away with them, but become harder to extend and compose
        - Could look something like this…

```
type Shape = Point -> Bool

inside:: Point -> Drawing -> Bool
```

For shapes:

```
circle = \(Vector x y) -> x ^ 2 + y ^ 2 <= 1
```

Transformations apply themselves to points, for example:

```
translate (Vector tx ty) = \(Vector px py) = Vector (px - tx) (py - ty)
```

(aside: why is this subtracting? We are applying the *inverse* of the transformation, because we are translating the point we are asking about)

Our interface

```
inside1 :: Point -> (Transform,Shape) -> Bool
inside1 p (t,s) = s . t p

inside :: Point -> Drawing -> Bool
inside p d = or $ map (inside1 p) d
```

    - **Deep Embedding**
        - Often more complex, but easier to optimise and add to
        - In a deep embedding types hold values
        - Images below show a deep embedding

```haskell
data Vector = Vector Double Double
type Point  = Vector

data Shape = Empty
           | Circle
           | Square
             deriving Show

empty = Empty
square = Square
circle = Circle

data Transform = Identity
           | Translate Vector
           | Scale Vector
           | Compose Transform Transform
           | Rotate Matrix
             deriving Show

data Matrix = Matrix Vector Vector
```

Some example transformation constructions:

```haskell
translate = Translate

rotate angle = Rotate $ matrix (cos angle) (-sin angle) (sin angle) (cos angle)
```

All the heavy lifting is done in the interpretation functions:

```haskell
transform :: Transform -> Point -> Point
transform (Translate (Vector tx ty)) (Vector px py)  = Vector (px - tx) (py - ty)
transform (Rotate m)                 p = (invert m) `mult` p

invert :: Matrix -> Matrix
mult :: Matrix -> Vector -> Vector

inside :: Point -> Drawing -> Bool
inside p d = or $ map (inside1 p) d

inside1 :: Point -> (Transform, Shape) -> Bool
inside1 p (t,s) = insides (transform t p) s

insides :: Point -> Shape -> Bool
p `insides` Empty = False
p `insides` Circle = distance p <= 1
p `insides` Square = maxnorm  p <= 1
```