COM2001 Advanced Programming Topics

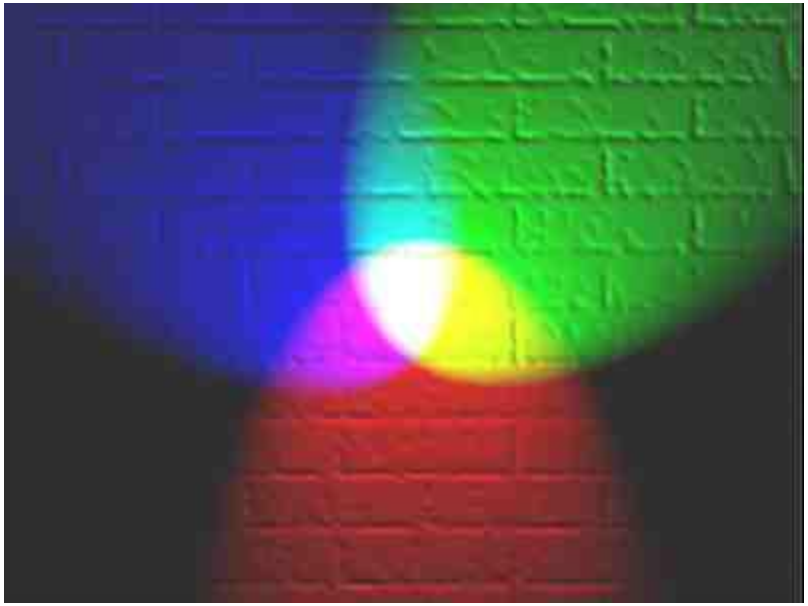
Using Haskell's class system

Algebraic Data Types

- There appear to be many sorts of algebraic data type in Haskell

data RGB = Red | Green | Blue

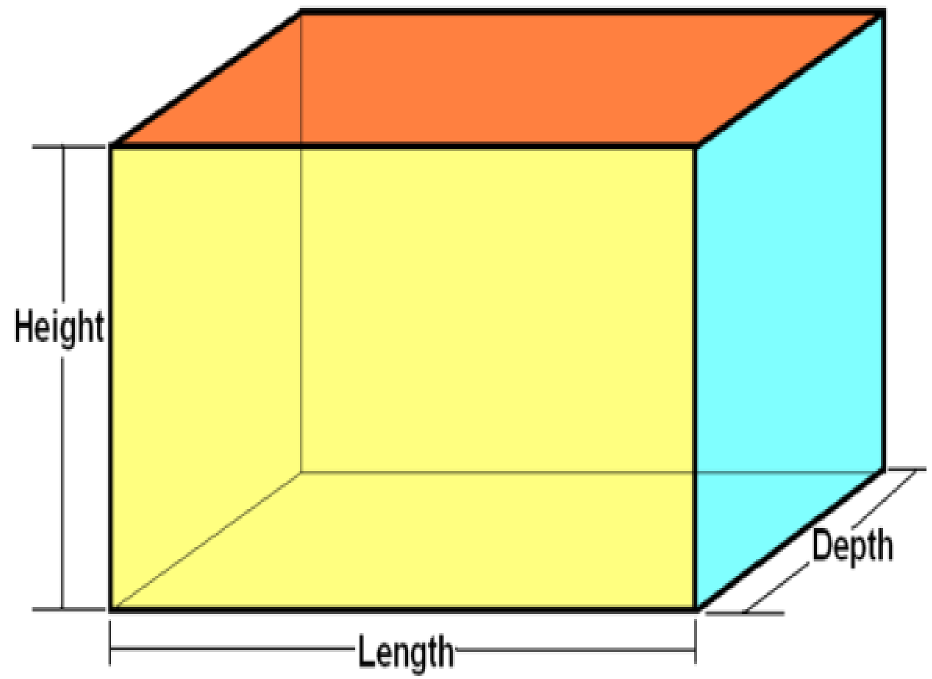
- The members of this type are simply listed one after the other
- The terms Red, Green and Blue are **nullary constructors**



http://en.wikipedia.org/wiki/RGB_color_model

data Block = Sides Float Float Float

- This type only has one constructor, which is applied to three arguments of type Float



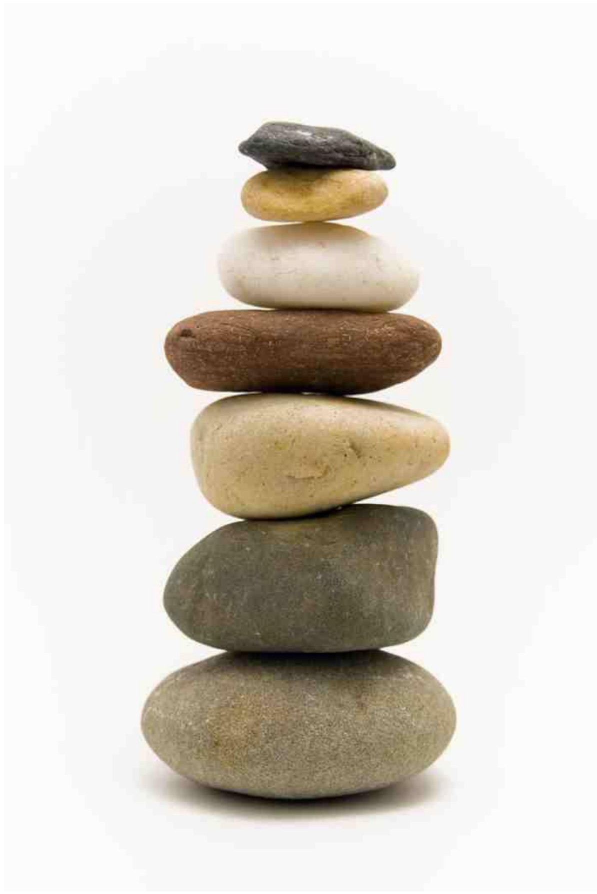
http://en.wikipedia.org/wiki/File:Cuboid.png

data Encapulate a = Enc a

- This type is **polymorphic**
- **Enc** will accept an argument of any type.

data Stack a = Empty | Push a (Stack a)

- This is a **recursive type**. The type we're defining appears in the definition itself.
- In this example, it is also a polymorphic type



Algebraic Data Types

- There appear to be many sorts of algebraic data type in Haskell
- … but Haskell treats them all the same way

Constructor syntax

- Constructors must start with a capital letter
- Type variables must start with a lower-case letter.
  - Stack a
  - Stack entries
  - Stack things

Haskell constructors

- You can't use the same constructor name more than once in the same scope
  - Empty :: Stack a
  - Empty :: Tree a
  - Empty :: Enc a

Types and constructors

- Types and constructors belong to different name spaces, so it's OK (but confusing) to use the same word as both a type name and a constructor name

```
type Name = String
type Phone = Int
type URL = String
data Person
    = Person Name Phone URL
        deriving (Eq, Show)
```

deriving

- Make the type a member of one or more **type classes**, and define (very basic) versions of associated functions

```
data Person …
  deriving (Eq, Show)

(==) :: Person -> Person -> Bool
show :: Person -> String
```

Accessing your data

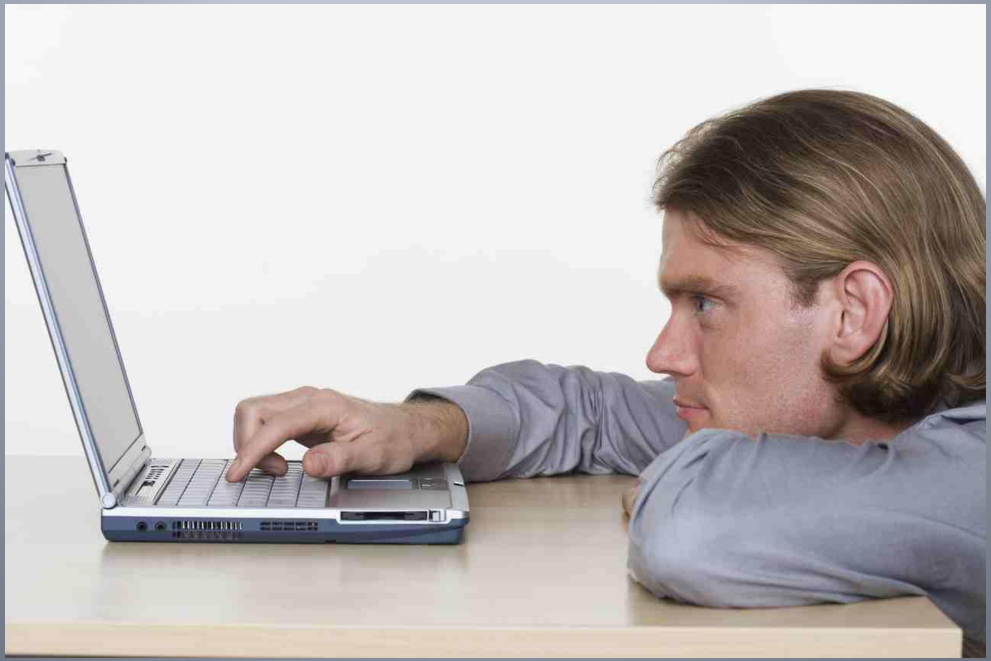
```
data Person = Person Name Phone URL
     deriving (Eq, Show)

name  :: Person -> Name
phone :: Person -> Phone
url   :: Person -> URL

name  (Person n _ _) = n
phone (Person _ p _) = p
url   (Person _ _ u) = u
```

Field label syntax

```
data Person = Person {
   name  :: Name,
   phone :: Phone,
   url   :: URL
 } deriving (Eq, Show)
```

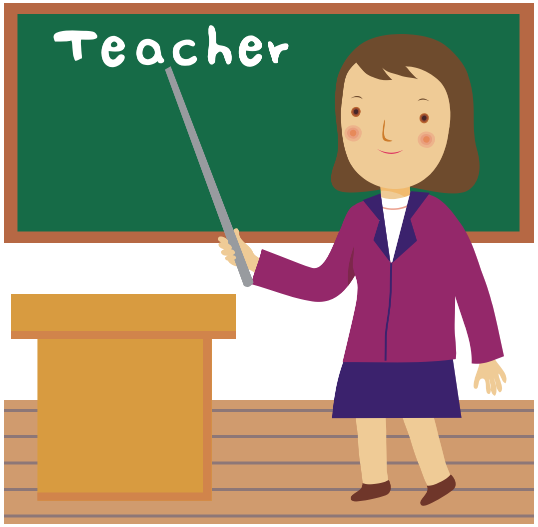
```
mps     = Person "mike" 2221800 "dcs"
myUrl   = url mps
newmps  = oldmps{phone = 2221841}
```

Type classes

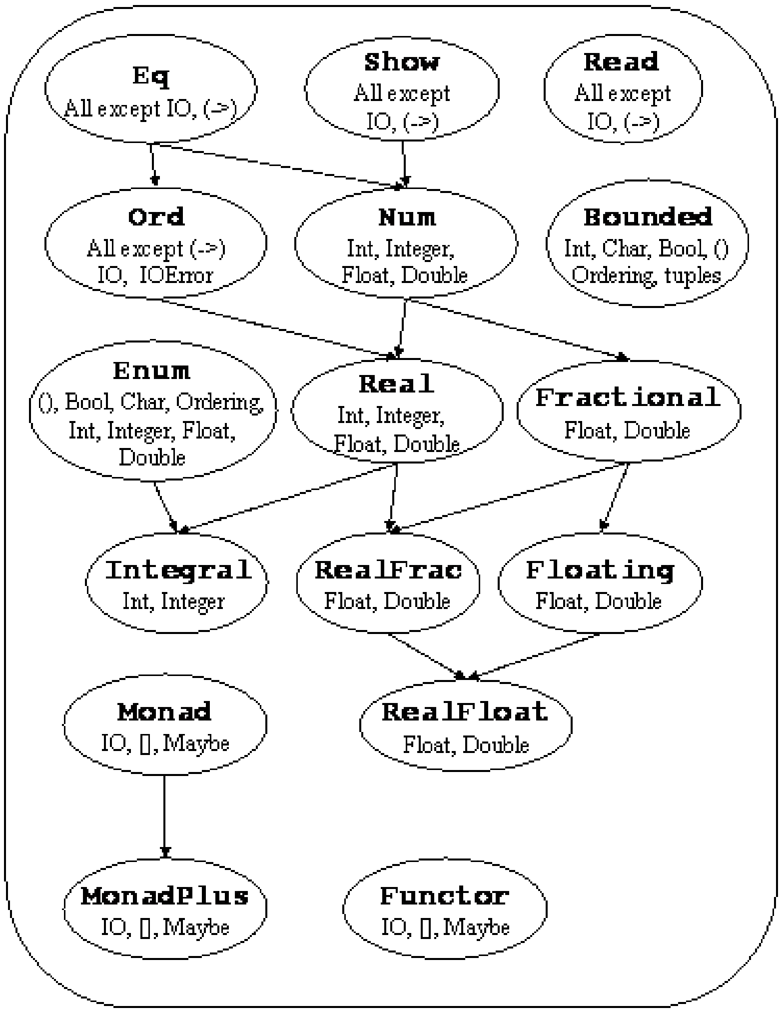


What is a type class?

- A collection of types, all of which have instances of specified polymorphic functions defined on them
- Defined using the **class** keyword, and then listing the relevant polymorphic functions



Standard Haskell Classes

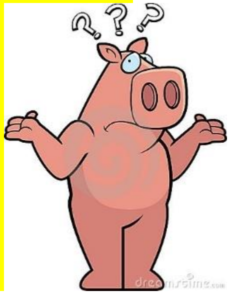


https://www.haskell.org/onlinereport/basic.html

Example: Eq

“To make a a member of Eq you need to define (==) and (/=) for it"

```
class Eq a where
  (==), (/=) :: a -> a -> Bool

  -- Minimal complete definition:
  --      (==) or (/=)

  -- Default definitions:
  x == y = not (x/=y)
  x /= y = not (x==y)
```

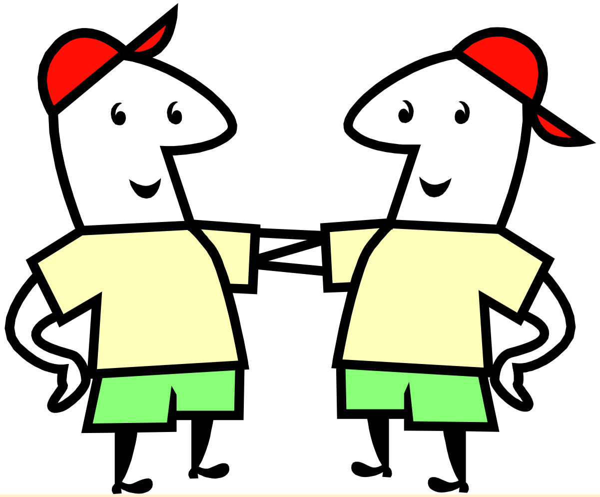


Never terminates?!

Assigning types to classes

```
instance Eq Person where
  -- specific definition of (==)
  p == p'  =  (name p) == (name p')
```

- Use the **instance** keyword
- Define the required functions



"Make the type Person an instance (i.e. member) of the class Eq"

Example: Show

```
class Show a where
  show :: a -> String
  ...

  -- Minimal complete definition:
  --     show or ...

  -- Default definitions
  ...
```

Why bother? Because ...

```
data List a = Singleton a
            | Join (List a) (List a)
              deriving Show

slogan :: List String
slogan = Join (Singleton "maths")
           (Join (Singleton "is") (Singleton "fun"))
```

Default version of **show slogan** using **deriving Show**

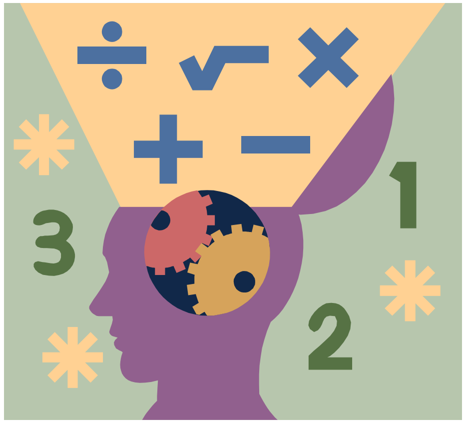
```
Join (Singleton "maths") (Join (Singleton "is") (Singleton "fun"))
```

... we can define our own output

```
instance (Show a) => Show (List a) where
  show (Singleton x) = show x
  show (Join xs ys)
          =  (show xs) ++ "  " ++ (show ys)
```

show slogan using **instance**

"maths is fun”



Read

```
instance (Read a) => Read (List a) where …
   …

input :: List String
input = read "maths is fun"
```

We'll see later how to parse input text

Constraints

```
instance (Show a) => Show (List a) where
  show (Singleton x) = show x
```

- **List a**'s definition of **show** relies on **show x** being defined for **x :: a**
- "**Provided a is in Show**, declare **List a** to be in **Show**, and define *show x* as specified"

Example: Container types

Stacks, lists, queues, …

- Stacks, lists and queues are all container types
  - They contain a collection of items
  - You can insert new items
  - You can remove items
  - You can test whether an item is in the container

Container types

- Some of the data types we're interested in:
  - Stack a – a stack containing objects of type a
  - List a – a list containing objects of type a
  - Stream a – a stream containing ….
  - Tape a – a tape containing objects ("symbols") of type a

In each case the type is of the form

ContainerName a

Container types

```
class Container c where
    create :: c a
    insert :: c a -> a -> c a
    contains :: c a -> a -> Bool
    remove :: c a -> a -> c a
    isEmpty :: c a -> Bool

instance Container Stack where …
instance Container Tape where …
```

Example: Stack

```
class Container c where
        create :: c a
        insert :: c a -> a -> c a
        contains :: c a -> a -> Bool
        remove :: c a -> a -> c a
        isEmpty :: c a -> Bool

instance Container Stack where
        -- create :: Stack a
        -- insert :: Stack a -> a -> Stack a
        -- contains :: Stack a -> a -> Bool
        -- remove :: Stack a -> a -> Stack a
        -- isEmpty :: Stack a -> Bool
```

The instance types are defined automatically. It's an error to define them again. You just need to implement the functions.

Example: [a]

```
class Container c where
        create :: c a
        insert :: c a -> a -> c a
        contains :: c a -> a -> Bool
        remove :: c a -> a -> c a
        isEmpty :: c a -> Bool

instance Container [] where
        create = []
        insert xs x = (x:xs)
        contains xs x = ( x `elem` xs)
        remove xs x = …
        isEmpty xs = …
```

data [] a = []
| (:) a ([] a)

[a] is a more familiar way of writing [] a

x:xs is the same as (:) x xs

Define each of the member functions

Pointwise equality…

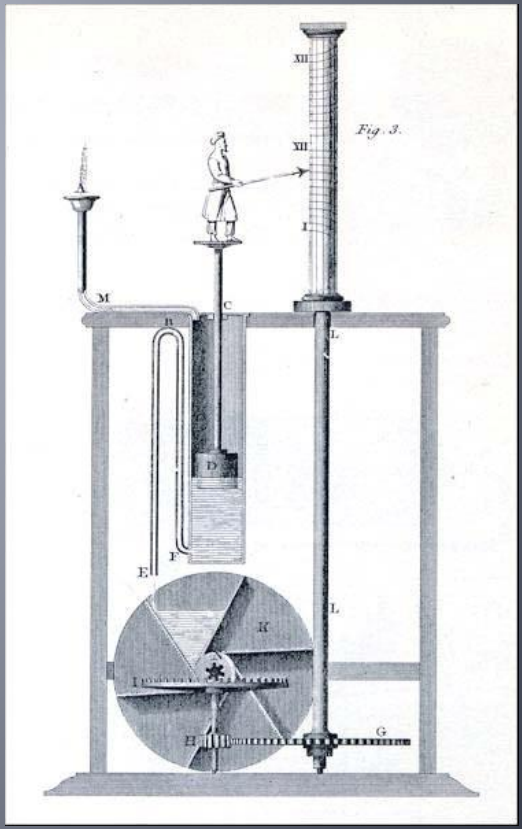
```
instance (Eq a) => Eq [a] where
        [] == []  = True
        [] == _   = False
        _  == []  = False
        (x:xs) == (y:ys) = (x==y) && (xs==ys)
```

Not matter what type a happens to be, if it's in the class Eq, then the type [a] can also be made into a member of Eq. Two lists are equal if they contains the same values in the same order.

Classes in action: automata

Warning – this is quite a complicated example. Take your time studying it!

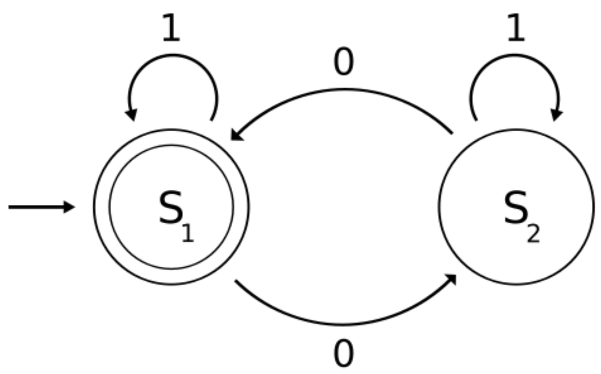
http://en.wikipedia.org/wiki/Automaton_clock



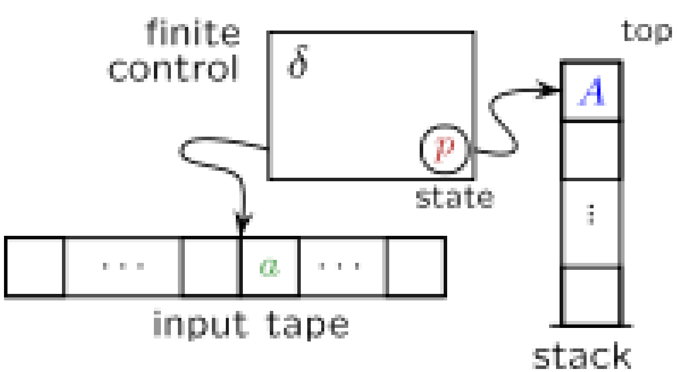
Modelling computation

Any state-machine model of computation can be described by specifying

- Its structure
- Its initial configuration (including its input)
- Which configurations are valid places to halt and/or accept inputs
- How it moves from one configuration to the next



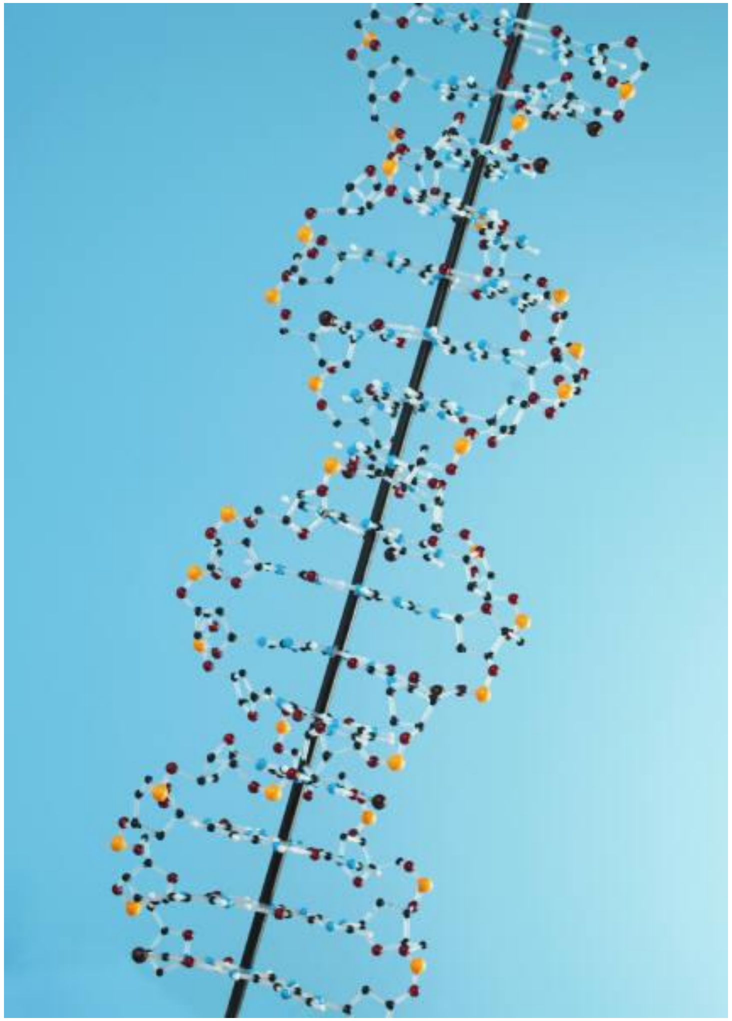
http://en.wikipedia.org/wiki/Finite-state_machine



http://en.wikipedia.org/wiki/Pushdown_automaton

Model

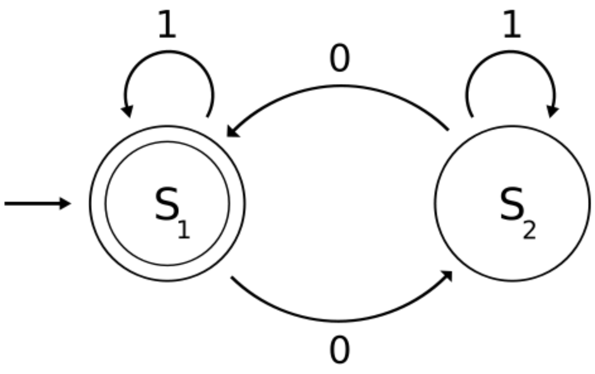
```
class (Eq cfg) => Model cfg where
  initialise :: String -> cfg
  acceptState :: cfg -> Bool
  doNextMove :: cfg -> cfg
  runFrom :: cfg -> cfg
  runModel :: String -> cfg

  -- Default implementation
  runModel = runFrom . initialise
```



Example: Finite state machine

- Suppose **s** is a finite set
- What do we need to add to **s** for it to be the underling state set for a FSM?

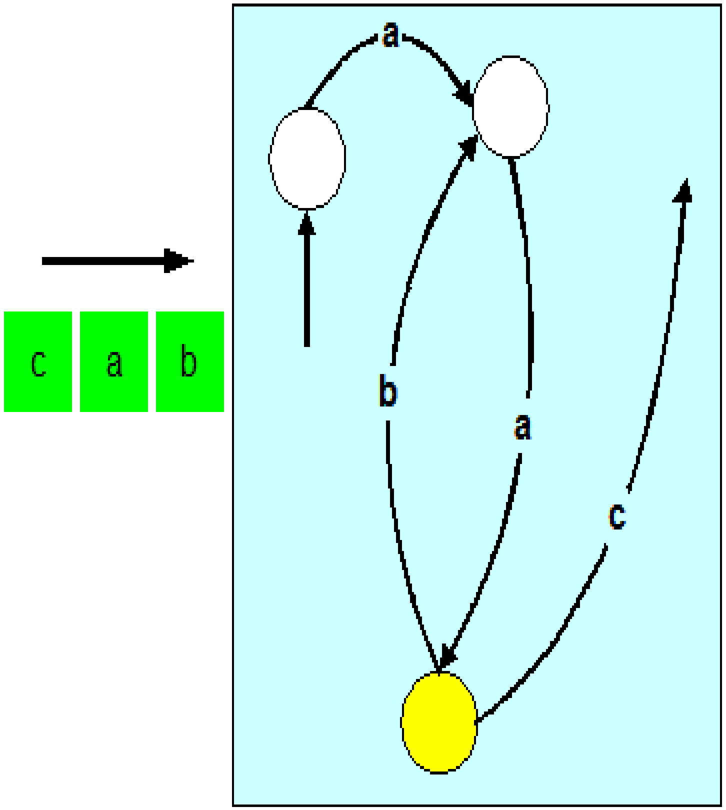
```
type Transitions s= [(s, Char, s)]

class (Eq s) => FSM s where
  transitions :: Transitions s
  initialState :: s
  haltStates :: [s]
```



http://en.wikipedia.org/wiki/Finite-state_machine

FSM configurations

```
data FSMConfig s
    = FSMConfig {
        state :: s,
        input :: String
      } deriving (Eq, Show)
```



- The current state
- The current input

Making FSM a model

```
class (Eq cfg) => Model cfg where
    initialise :: String -> cfg
    acceptState :: cfg -> Bool
    doNextMove :: cfg -> cfg
    runFrom :: cfg -> cfg
    runModel :: String -> cfg
```

```
class FSM s where
 transitions :: Transitions s
 initialState :: s
 haltStates :: [s]

data FSMConfig s =
            FSMConfig s String
```

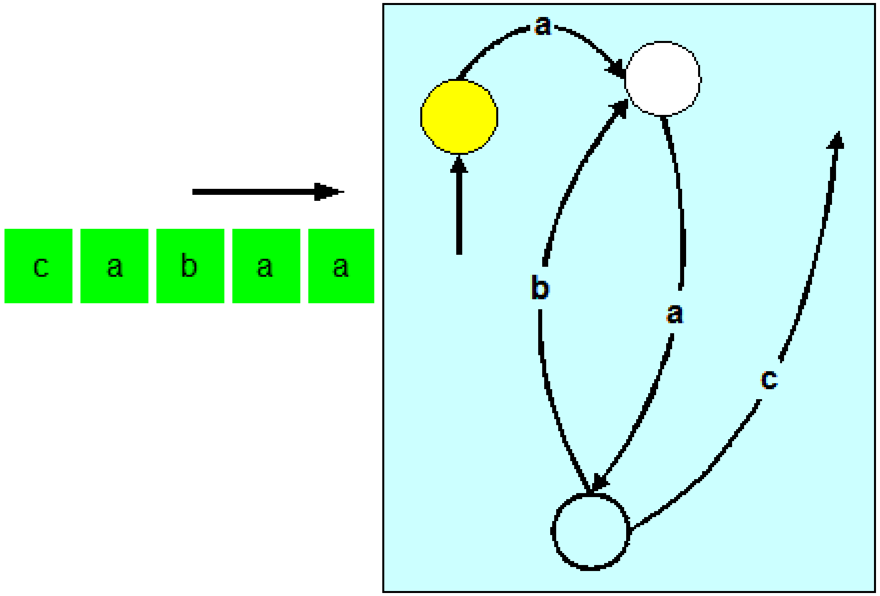
```
instance (FSM s)  => Model (FSMConfig s)
```

FSMConfig s is a set of FSM configurations. They can be used as the configurations of a **Model** provided we have the functions available that are defined in the class **FSM**

initialise

initialise :: String -> **(FSMConfig s)**
initialise str = FSMConfig **initialState** str

```
class FSM s where
 transitions :: Transitions s
 initialState :: s
 haltStates :: [s]

data FSMConfig s =
          FSMConfig s String
```



acceptState

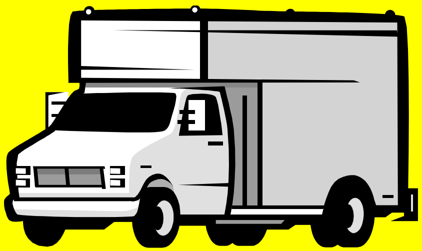
```
acceptState :: (FSMConfig s) -> Bool
acceptState (FSMConfig s i)
            = (s `elem` haltStates) && (null i)
```


```
class FSM s where
 transitions :: Transitions s
 initialState :: s
 haltStates :: [s]

data FSMConfig s =
          FSMConfig s String
```

The machine has reached a valid halt state and has no more inputs to process

doNextMove

```
doNextMove :: (FSMConfig s) -> (FSMConfig s)
doNextMove cfg@(FSMConfig s i)
  | null i = cfg
  | otherwise = FSMConfig nextstate (tail i)
    where nextstate = head nextstates
          nextstates = [s' | (s, c, s') <- transitions, c == head i]
```

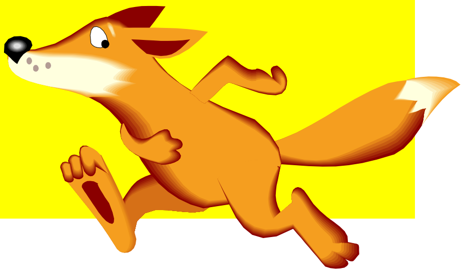


```
class FSM s where
 transitions :: Transitions s
 initialState :: s
 haltStates :: [s]
```

```
data FSMConfig s =
        FSMConfig s String
```

runFrom

```
runFrom :: (FSMConfig s) -> (FSMConfig s)
runFrom cfg@(FSMConfig s i)
    | null i = cfg
    | acceptState cfg = cfg
    | otherwise = runFrom (doNextMove cfg)
```



```
class FSM s where
 transitions :: Transitions s
 initialState :: s
 haltStates :: [s]
```

```
data FSMConfig s =
            FSMConfig s String
```

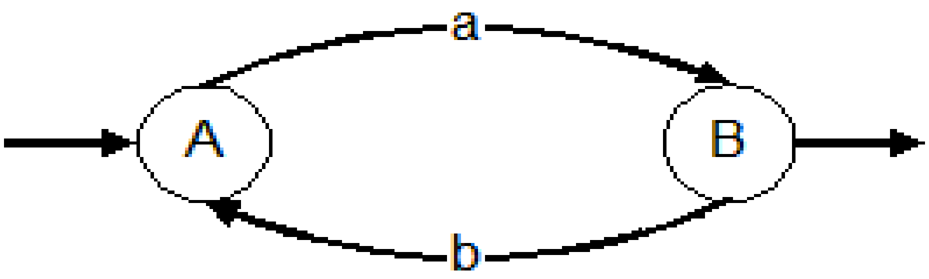
Summary so far

We've

- assumed that **s** is in the class **FSM**
- defined **FSMConfig s**
- added **FSMConfig s** to the class **Model**

- All that remains is to say what **s** is.
- This depends on the actual machine being defined.

Declaring a specific FSM




```
data MyStates = A | B deriving (Eq, Show)

instance FSM MyStates where
  transitions  = [(A, 'a', B),
                  (B, 'b', A)]
  initialState = A
  haltStates   = [B]
```

Recognising strings?

- Given str :: String
- Run a machine on str
- Report whether we reached an acceptState


```
recognises :: String -> Bool
recognises str
  = acceptState (runModel str)
```

This fails. We haven't said which machine to use!

What went wrong?

```
recognises :: String -> Bool
recognises str = acceptState (runModel str)
```

```
runModel    :: Model cfg => String -> cfg
acceptState :: Model cfg =>           cfg -> Bool
```

```
recognises  :: Model cfg => String        -> Bool
```

recognises can’t work out what type cfg actually is. It doesn’t know what machine to use.

Recognising strings!

- Given **str** :: String
- Run <u>this</u> machine on **str**
- Report whether we reached an **acceptState**

```
data S = A | B deriving (Eq, Show)

recognises :: String -> Bool
recognises str =
  acceptState (runModel str :: FSMConfig S)
```

SUMMARY

- **Classes** - collect types with specific functions defined on them
- **Constraints** - some functions rely on their arguments belonging to specific classes
- **Using classes** - how to define all state-based computational models at the same time