Haskell Overview IV (4A)

Young Won Lim
11/4/16

Copyright (c) 2016 Young W. Lim.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using OpenOffice.

Based on

Haskell Tutorial, Medak & Navratil
ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf

Yet Another Haskell Tutorial, Daume
https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf

Class Constraint : (Eq a) =>

***Context** (Eq a)*

instance (Eq a) => Eq (Tree a) where …

the type a is *constrained*
by the context (Eq a)

the **types** of a must *belong*
to the Eq **type class**

the **types** of **a** must *implement*
operations **==** and ***/=***

( ) pattern matching

https://www.haskell.org/tutorial/stdclasses.html

Eq Instance : Eq (Tree a)

Eq Instance Eq (Tree a)

( ) pattern matching

instance (Eq a) => Eq (Tree a) where …

type **instance**

The **type** **Tree** **a** is an **instance** of the **class** **Eq**,
whose **method** == and /= are defined

https://www.haskell.org/tutorial/stdclasses.html

Eq Instance Declaration of Tree Type

Eq Instance Eq (Tree a)

```
instance  (Eq a) => Eq (Tree a)  where
    (Leaf       x) == (Leaf       y)  =   x == y
    (Branch l r)  == (Branch l' r')   =   l == l' && r == r'
    _              == _               =   False
```

(Eq a) =>

Eq (Tree a)

( ) pattern matching

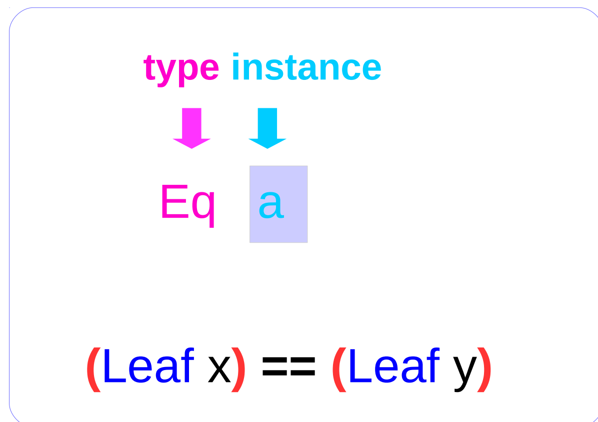
https://www.haskell.org/tutorial/stdclasses.html

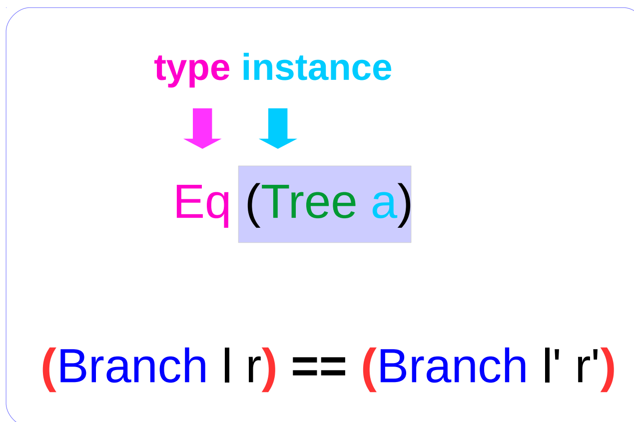
Two types of Equality : constrained instance

Eq Instance Eq (Tree a)

( ) pattern matching

```
instance (Eq a) => Eq (Tree a) where
    (Leaf x)     == (Leaf y)       =   x == y
    (Branch l r) == (Branch l' r') =   l == l' && r == r'
    _            == _              =   False
```





https://www.haskell.org/tutorial/stdclasses.html

Two types of Equality: using derived instances

*Automatically Derived **Eq** **Instance***

```
data  Tree a          =  Leaf a | Branch (Tree a) (Tree a)  deriving Eq
```

deriving Eq : implicitly produces Eq instance declarations

Eq a   Eq (Tree a)   **( ) pattern matching**

https://www.haskell.org/tutorial/stdclasses.html

Derived Instances

the following instances can be derived automatically from the **data** declaration

data	**...**	**=**	**...**	**deriving** Ord
data	**...**	**=**	**...**	**deriving** Enum
data	**...**	**=**	**...**	**deriving** Ix
data	**...**	**=**	**...**	**deriving** Read
data	**...**	**=**	**...**	**deriving** Show

https://www.haskell.org/tutorial/stdclasses.html

Multiple Derived Instances

```
instance (Eq a) => Eq (Tree a) where
    (Leaf x)       == (Leaf y)        =  x == y
    (Branch l r)   == (Branch l' r')  =  l == l' && r == r'
    _              == _               =  False

instance (Ord a) => Ord (Tree a) where
    (Leaf _)       <= (Branch _)      =  True
    (Leaf x)       == (Leaf y)        =  x <= y
    (Branch _)     <= (Leaf _)        =  False
    (Branch l r)   == (Branch l' r')  =  l == l' && r <= r' || l <= l'

data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving (Eq, Ord)

data [a] = [] | a : [a] deriving (Eq, Ord)
```

https://www.haskell.org/tutorial/stdclasses.html

Extension to Derived Instances

```
data T0 f a =    MkT0 a           deriving ( Eq )
data T1 f a =    MkT1 (f a)       deriving ( Eq )
data T2 f a =    MkT2 (f (f a))   deriving ( Eq )    X
```

```
instance Eq a        =>    Eq (T0 f a) where ...
instance Eq (f a)    =>    Eq (T1 f a) where ...
instance Eq (f (f a)) =>   Eq (T2 f a) where …    X
```

GHC accepts the first two, but not the third.

each constraint in the inferred instance context must
consist only of type variables, with no repetitions.

https://downloads.haskell.org/~ghc/7.8.4/docs/html/users_guide/deriving.html

Multi-parameter Type Class Definition

type class Eq

type a

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

SPTC (Single Parameter Type Class) :
a type class is a set of types

MPTC (Multi Parameter Type Class):
a type class is a relation between types

```
class Monad m => VarMonad m v where
  new    ::   a   -> m (v a)
  get    :: v a  -> m a
  put    :: v a  -> a -> m ()
```

( ) pattern matching

https://wiki.haskell.org/Multi-parameter_type_class

Multi-parameter Type Instance Declaration

```
class Monad m => VarMonad m v where
  new     ::   a   -> m (v a)
  get     :: v a  -> m a
  put     :: v a  -> a -> m ( )
```

```
instance VarMonad IO IORef where ...
```

```
instance VarMonad (ST s) (STRef s) where ...
```

( ) pattern matching

https://wiki.haskell.org/Multi-parameter_type_class

Multi-parameter Type Class Definition

```
class Collection c a where
  union :: c a -> c a -> c a
  ...etc.
```

https://wiki.haskell.org/Multi-parameter_type_class

Multi-parameter Type Class Definition

```
class Functor (m k) => FiniteMap m k where
  ...

class (Monad m, Monad (t m)) => Transform t m where
  lift :: m a -> (t m) a

class C a where {
  op :: D b => a -> b -> b
}
```

https://wiki.haskell.org/Multi-parameter_type_class

Multi-parameter Type Class Definition

```
class C a => D a where { ... }

class A cls c where
  meth :: cls c => c -> c

class A B c => B c where
```

https://wiki.haskell.org/Multi-parameter_type_class

Multi-parameter Type Class Definition

```
class Monad m => VarMonad m v where
  new    ::   a   -> m (v a)
  get    :: v a  -> m a
  put    :: v a  -> a -> m ( )
```

```
instance VarMonad IO       IORef       where ...
instance VarMonad (ST s) (STRef s)     where …
```

{-# LANGUAGE MultiParamTypeClasses #-} pragma

https://wiki.haskell.org/Multi-parameter_type_class

Object Instance of **Objects** Type Class

```
type ID = Int
type Attrib = (String, String)

class Objects o where
     object     :: ID -> [Attrib] -> o
     getID      :: o -> ID
     getAttr    :: o -> [Attrib]
     getName    :: o -> String
     getName = snd . head . filter ((“name”==) . fst) . GetAttr

data Object = Obj ID [Attrib] deriving Show

instance Objects Object
     object   i as          = Obj i as
     getID    (Obj i as)    = i
     getAtts  (Obj i as)    = as
     getName = snd . head . filter ((“name”==) . fst) . GetAttr
```

Object ➡ o

The **type** o belongs to the **type class** Objects.
The **type** o must implement **object**, **getID**, **getAttr**, **getName** methods

The **data constructor** of the **type** Object

The **type** Object is an **instance** of the **type class** Objects.

DBS Object Instance of Databases Type Class

```
type ID = Int
type Attrib = (String, String)

class (Objects o) => Databases d o where
      empty        :: d o
      getLastID    :: d o -> ID
      getObjects   :: d o -> [o]
      setLastID    :: ID -> d o -> d o
      setObjects   :: [o] -> d o -> d o
      . . .

data DBS o = DB ID [o] deriving Show

instance Databases DBS Object where
      empty        =      DB 0 []
      getLastID          (DB i os) = i
      setLastID    i     (DB j os) = DB i os
      getObjects         (DB i os) = os
      setObjects   os    (DB i ps) = DB i os
```

Object ➡ o

The **type** d o belongs to the **type class** Databases.
The **type** d o must implement **get/setLastID**, **get/setObjects**, **insert, select, selectBy** methods

The **type** o belongs to the **type class** Objects.
The **type** o must implement **object**, **getID**, **getAttr**, **getName** methods

The **data constructor** of the **type** DBS o

The **type** DBS Object is an **instance** of the **type class** Databases.

A Simple Database

```
type ID = Int
type Attrib = (String, String)
```

```
class (Objects o) => Databases d o where
    empty       :: d o
    getLastID   :: d o -> ID
    getObjects  :: d o -> [o]
    setLastID   :: ID -> d o -> d o
    setObjects  :: [o] -> d o -> d o
    . . .

data DBS o = DB ID [o] deriving Show

instance Databases DBS Object where
    Empty                        = DB 0 []
    getLastID        (DB i os) = i
    setLastID   i    (DB j os) = DB i os
    getObjects       (DB i os) = os
    setObjects  os   (DB i ps) = DB i os
```

```
class Objects o where
    object      :: ID -> [Attrib] -> o
    getID       :: o -> ID
    getAttr     :: o -> [Attrib]
    getName     :: o -> String
    GetName =   …

data Object = Obj ID [Attrib] deriving Show

instance Objects Object
    object  i as          = Obj i as
    getID   (Obj i as)    = i
    getAtts (Obj i as)    = as
```

A Simple Database – data constructors

```
type ID = Int
type Attrib = (String, String)
```

```
class (Objects o) => Databases d o
      . . .

data DBS o = DB ID [o]

instance Databases DBS Object
      . . .
```

```
class Objects o where
      . . .

data Object = Obj ID [Attrib]

instance Objects Object
      . . .
```

The **type DBS Object** is an **instance** of the **type class** Databases.

The **type Object** is an **instance** of the **type class** Objects.

DB ID [Object] is the **data constructor** of the **type DBS Object**

Obj ID [Attrib] is the **data constructor** of the **type Object**

[Object] a list of Object

[Attrib] a list of Attrib

Databases Type Class Definition

```
class (Objects o) => Databases d o where
      empty         :: d o
      getLastID     :: d o -> ID
      getObjects    :: d o -> [o]
      setLastID     :: ID -> d o -> d o
      setObjects    :: [o] -> d o -> d o

      insert :: [Attrib] -> d o -> d o
      insert as db = setLastID i' db' where
                     db' = setObjects os' db
                     os' = o : os
                     os  = getObjects db
                     o   = object i' as
                     i'  = 1 + getLastID db

      select :: ID ->  d o -> o
      select i = head . filter ((i==).getID) . GetObjects

      selectBy :: (o -> Bool) -> d o ->  [o]
      selectBy f = filter f . getObjects
```

The **type** d o belongs to the **type class** Databases.
The **type** d o must implement **get/setLastID**, **get/setObjects**, **insert, select, selectBy** methods

The **type** o belongs to the **type class** Objects.
The **type** o must implement **object**, **getID**, **getAttr**, **getName** methods

prepend operator :
x : xs

Databases Type Class Definition - rearranged

```
class (Objects o) => Databases d o where
      empty          :: d o
      getLastID      :: d o -> ID
      getObjects     :: d o -> [o]
      setLastID      :: ID -> d o -> d o
      setObjects     :: [o] -> d o -> d o

      insert :: [Attrib] -> d o -> d o
      select :: ID ->  d o -> o
      selectBy :: (o -> Bool) -> d o ->  [o]

      insert as db = setLastID i' db' where
                   db' = setObjects os' db
                   os' = o : os
                   os  = getObjects db
                   o   = object i' as
                   i'  = 1 + getLastID db

      select i = head . filter ((i==).getID) . GetObjects
      selectBy f = filter f . getObject
```

The **type** d o belongs to the **type class** Databases.
The **type** d o must implement **get/setLastID**, **get/setObjects**, **insert, select, selectBy** methods

The **type** o belongs to the **type class** Objects.
The **type** o must implement **object**, **getID**, **getAttr**, **getName** methods

prepend operator **:**
x : xs

default methods

DBS Object : Instance and Data Constructor

```
class (Objects o) => Databases d o where …

data DBS o = DB ID [o] deriving Show

instance Databases DBS Object where …
```

type class	instance type
Databases	DBS Object

DBS Object is a type of Databases type class
A parameterized type (Object: a parameter)

type constructor		data constructor
DBS Object	=	DB ID [Object]

DBS Object type variable can be constructed by
DB ID [Object]

DBS Object : Insert method – Type Signature

```
class (Objects o) => Databases d o where …

data DBS o = DB ID [o] deriving Show

instance Databases DBS Object where …
```

```
insert :: [Attrib] -> d o -> d o
```

DBS Object is a type of Databases type class,
a parameterized type (Object: a parameter)

Attribute list

```
d0, d1, d2 :: DBS Object
d0 = empty
d1 = insert [("name", "john"), ("age", "30")]  d0
d2 = insert [("name", "mary"), ("age", "20")]  d1
     insert as                                  db
```

```
type ID = Int
type Attrib = (String, String)
```

DBS Object : Insert method – Definition

```
class (Objects o) => Databases d o where …

data DBS o = DB ID [o] deriving Show

instance Databases DBS Object where …
```

```
type ID = Int
type Attrib = (String, String)
```

d1 = **insert** [(“name”, “john”), (“age”, “30”)] d0

insert as db = **setLastID** i’ db’ where . . .

db : DBS Object type

as : **[Attrb]** type

Method **insert** Definition – creating an Object

```
class Objects o where
    object    :: ID -> [Attrib] -> o
    getID     :: o -> ID
    getAttr   :: o -> [Attrib]
    getName   :: o -> String
    GetName = …

data Object = Obj ID [Attrib] deriving Show

instance Objects Object
    object  i as          = Obj i as
    getID   (Obj i as)    = i
    getAtts (Obj i as)    = as
```

```
insert :: [Attrib] -> d o -> d o
insert as db = setLastID i' db' where
```

as db

```
i' = 1 + getLastID db      <- db
o = object i' as           <- as
```

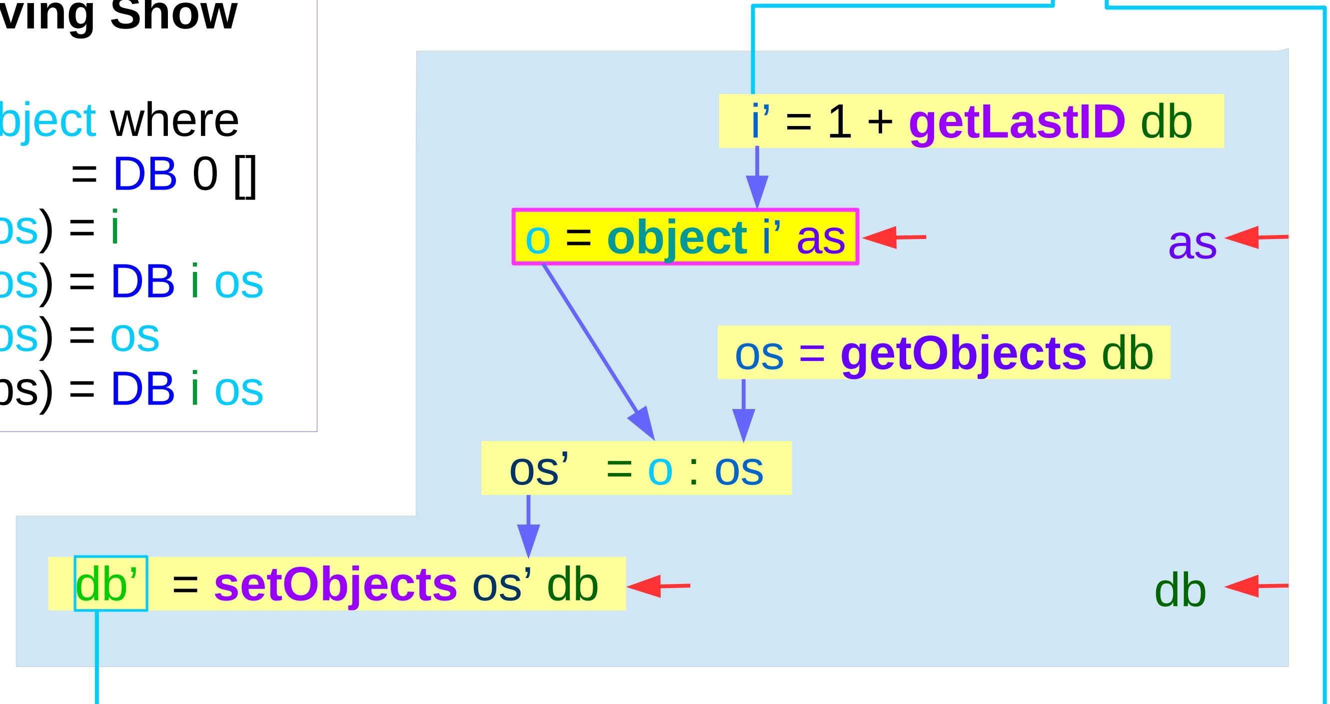
```
d1 = insert [("name", "john"), ("age", "30")] d0
```

```
o      Obj i' [("name", "john"), ("age", "30")]
```

Method **insert** Definition – updating DBS Object

```
type ID = Int
type Attrib = (String, String)
```

```
class (Objects o) => Databases d o
      . . .

data DBS o = DB ID [o] deriving Show

instance Databases DBS Object where
      Empty                       = DB 0 []
      getLastID       (DB i os) = i
      setLastID    i  (DB j os) = DB i os
      getObjects      (DB i os) = os
      setObjects  os  (DB i ps) = DB i os
```

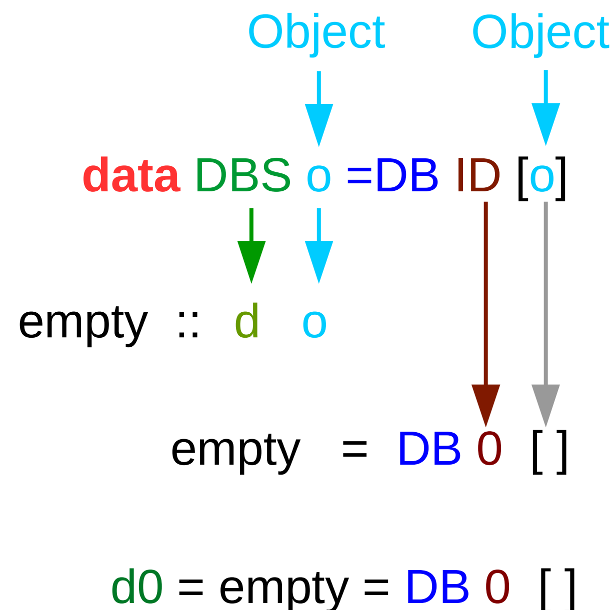


Testing the Database

```
d0, d1, d2 :: DBS Object
d0 = empty
d1 = insert [("name", "john"), ("age", "30")]    d0
d2 = insert [("name", "mary"), ("age", "20")]   d1


test1 :: Object
test1 = select 1 d1
test2 :: Object
test2 = selectBy (("john" ==) . getName) d2
```

DBS Object : empty variable

```
class (Objects o) => Databases d o where
      empty    :: d o
      …

data DBS o = DB ID [o] deriving Show

instance Databases DBS Object where
      empty   =  DB 0 []
      …
```




```
d0, d1, d2 :: DBS Object
d0 = empty
d1 = insert [(“name”, “john”), (“age”, “30”)]   d0
d2 = insert [(“name”, “mary”), (“age”, “20”)]  d1
     insert as                                  db
```

```
type ID = Int
type Attrib = (String, String)
```

Creating the 1st Object

```
class Objects o where
      object        :: ID -> [Attrib] -> o
      …

data Object = Obj ID [Attrib] deriving Show

instance Objects Object
      object   i as  = Obj i as
      …
```

```
d0, d1, d2 :: DBS Object

d0 = DB 0 []

d1 = insert [("name", "john"), ("age", "30")] d0
```

Arrows point from `[("name", "john"), ("age", "30")]` to **as** and from `d0` to **db**.

```
i' = 1 + getLastID d0  = 1
o = object i' as
  =  object 1 [("name", "john"), ("age", "30")]
  =  Obj 1 [("name", "john"), ("age", "30")]
```

d1 : Updating a Databases d0

```
class (Objects o) => Databases d o where
      empty          :: d o
      …
      insert :: [Attrib] -> d o -> d o
      insert as db = setLastID i' db' where
                 db'  = setObjects os' db
                 os'  = o : os
                 os   = getObjects db
                 o    = object i' as
                 i'   = 1 + getLastID db
      …

data DBS o = DB ID [o] deriving Show

instance Databases DBS Object where
      empty = DB 0 []
      …
```

```
d0, d1, d2 :: DBS Object

d0 = DB 0 []

d1 = insert [("name", "john"), ("age", "30")] d0
                  ↓                             ↓
                  as                            db

i' = 1 + getLastID d0   = 1
o = object i' as
  = object 1 [("name", "john"), ("age", "30")]
  = Obj 1 [("name", "john"), ("age", "30")]
os = getObjects d0 = getObjects DB 0 [] = []
os' = o : os
     = [ Obj 1 [("name", "john"), ("age", "30")]  ]
db' = setObjects os' d0
  = DB 0 [ Obj 1 [("name", "john"), ("age", "30")]  ]
setLastID i' db'
  = DB 1 [ Obj 1 [("name", "john"), ("age", "30")]  ]

d1 = DB 1 [ Obj 1 [("name", "john"), ("age", "30")]  ]
```

d2: Updating a Databases d1

```
d0 = DB 0 [ ]
d1 = DB 1 [ Obj 1 [(“name”, “john”), (“age”, “30”)]  ]

d2 = insert [(“name”, “mary”), (“age”, “20”)]    d1
                                            ⇓            ⇓
                                           as           db

i’ = 1 + getLastID d1   = 2
o = object i’ as
  = object 2 [(“name”, “mary”), (“age”, “20”)]
  = Obj 2 [(“name”, “mary”), (“age”, “20”)]
os = getObjects d1 = getObjects DB 1 [Obj 1 [(“name”, “john”), (“age”, “30”)] ] = [Obj 1 [(“name”, “john”), (“age”, “30”)]]
os’ = o : os
    = [ Obj 2 [(“name”, “mary”), (“age”, “20”)] , Obj 1 [(“name”, “john”), (“age”, “30”)] ]
db’ = setObjects os’ d1
  = DB 1 [ Obj 2 [(“name”, “mary”), (“age”, “20”)] , Obj 1 [(“name”, “john”), (“age”, “30”)] ]
setLastID i’ db’
  = DB 2 [ Obj 2 [(“name”, “mary”), (“age”, “20”)] , Obj 1 [(“name”, “john”), (“age”, “30”)] ]

d2  = DB 2 [ Obj 2 [(“name”, “mary”), (“age”, “20”)] , Obj 1 [(“name”, “john”), (“age”, “30”)] ]
```

Selecting the 1st Object d1

```
d0 = DB 0 [ ]
d1 = DB 1 [ Obj 1 [(“name”, “john”), (“age”, “30”)]  ]
d2 = DB 2 [ Obj 2 [(“name”, “mary”), (“age”, “20”)] , Obj 1 [(“name”, “john”), (“age”, “30”)] ]
```

```
select :: ID ->  d o -> o
select i = head . filter ((i==).getID) . getObjects
```

```
select 1   d1 = head . filter ((1==).getID) . getObjects  d1
```

```
test1 :: Object
test1 = select 1 d1
```

```
test2 :: Object
test2 = selectBy ((“john” ==).getName) d2
```

```
                           getObjects d1 = [ Obj 1 [(“name”, “john”), (“age”, “30”)]  ]
    filter ((1==).getID) . getObjects d1 = [ Obj 1 [(“name”, “john”), (“age”, “30”)]  ]
head . filter ((1==).getID) . getObjects d1 =  Obj 1 [(“name”, “john”), (“age”, “30”)]
```

Selecting by name

```
d0 = DB 0 [ ]
d1 = DB 1 [ Obj 1 [("name", "john"), ("age", "30")]  ]
d2 = DB 2 [ Obj 2 [("name", "mary"), ("age", "20")] , Obj 1 [("name", "john"), ("age", "30")] ]
```

```
selectBy :: (o -> Bool) -> d o ->  [o]
selectBy f = filter f . getObjects
```

```
selectBy f d1= filter f . getObjects d1
```

```
test2 :: Object
test2 = selectBy (("john" ==) . getName) d2
```

```
getObjects d2 = [ Obj 2 [("name", "mary"), ("age", "20")] , Obj 1 [("name", "john"), ("age", "30")] ]
```

```
selectBy (("john" ==).getName)  .  getObjects d2 = [ Obj 1 [("name", "john"), ("age", "30")]  ]
```