Cursul 8 - ADT

- Tipuri de date abstracte (ADT)
  - Modulul – mecanism pentru implementarea unui adt
  - Exemplul1: Tipul abstract Queue
  - Exemplul2: Tipul abstract Set
    - Specificare algebrică
    - Implementare

Module

- Modulul – mecanism pentru definirea unui adt
- Sintaxa:

```
module Nume_modul(Lista_export)where
        Implementare
```

- *Nume_modul* începe cu literă mare
- *Lista_export* conține:
  - Numele tipului abstract de date – acelaşi cu numele modulului
  - Numele operațiilor
  - Nici un alt nume sau valoare declarat în modul şi care nu apare în lista export nu poate fi utilizat în altă parte
  - Asta înseamnă că implementarea descrisă în modul este ascunsă în orice script ce foloseşte modulul

Exemplu: modulul Queue

```
module Queue(Queue, empty, isEmpty, join, front, back) where
newtype Queue a = MkQ([a], [a])
     deriving (Show)

empty :: Queue a
empty = MkQ([], [])

isEmpty :: Queue a -> Bool
isEmpty(MkQ(xs, ys)) = null xs

join :: a -> Queue a -> Queue a
join x (MkQ(ys,zs)) = mkValid(ys, x:zs)

front :: Queue a -> a
front(MkQ(x:xs, ys)) = x

back :: Queue a -> Queue a
back(MkQ(x:xs, ys)) = mkValid(xs, ys)

mkValid :: ([a], [a]) -> Queue a
mkValid (xs, ys) = if null xs then MkQ(reverse ys, [])
            else MkQ(xs, ys)
```

Module - utilizare

- Utilizarea unui modul într-un script se face folosind o declarație import în acel script:

  import *Nume_modul*

- Exemplu: scriptul coada.hs

```
import Queue
toQ :: [a] -> Queue a
toQ = foldr join empty.reverse
fromQ :: Queue a -> [a]
fromQ q = if isEmpty q then [] else front q:fromQ(back q)

c1 :: Queue Int
c2 :: Queue [Char]
c1 = join 1(join 2( join 3( join 4(join 5 empty))))
c2 = join "ion" (join "vasile"(join "ana" empty))
```

Exemple

```
Main> join 1(join 2(join 3 empty))
MkQ ([3],[1,2])
Main> c1
MkQ ([5],[1,2,3,4])
Main> c2
MkQ (["ana"],["ion","vasile"])
Main> toQ [1,2,3,4,5]
MkQ ([1],[5,4,3,2])
Main> toQ ['a','b','c']
MkQ ("a","cb")
Main> join 8 c1
MkQ ([5],[8,1,2,3,4])
Main> join "horia" c2
MkQ (["ana"],["horia","ion","vasile"])
Main> front c2
"ana"
Main> front (back c2)
"vasile"
```

Exemple

```
Main> mkValid ([1,2,3], [4,5])
ERROR - Undefined variable "mkValid"
-- daca adaug mkValid în lista_export
Main> :r
Main> mkValid ([1,2,3], [4,5])
MkQ ([1,2,3],[4,5])
Main> mkValid ([], [2,4,5])
MkQ ([5,4,2],[])
```

Mulțimi

- Mulțimile pot fi reprezentate prin:
  - Liste
  - Liste fără elemente duplicate
  - Liste ordonate
  - Arbori
  - Funcții booleene
  - etc.
- Reprezentarea este aleasă în funcție de operațiile ce se folosesc

Mulțimi

- Operații pe mulțimi:

```
empty    :: Set a
isEmpty  :: Set a -> Bool
member   :: Set a -> a -> Bool
insert   :: a -> Set a -> Set a
delete   :: a -> Set a -> Set a
```

- Un tip bazat pe cele 5 operații: dicționar

```
union :: Set a -> Set a -> Set a
meet  :: Set a -> Set a -> Set a
minus :: Set a -> Set a -> Set a
```

Mulțimi - specificarea algebrică

```
insert x (insert x xs) = insert x xs
insert x (insert y xs) = insert y (insert x xs)

isEmpty empty = True
isEmpty(insert x xs) = False

member empty y = False
member(insert x xs) y = (x == y) `or` member xs y

delete x empty = empty
delete x (insert y xs) = if x == y then delete x xs
                         else insert y (delete x xs)
```

Mulțimi - specificarea algebrică

```
union xs empty = xs
union xs (insert y ys) = insert y (union xs ys)

meet xs empty = empty
meet xs (insert y ys) =
    if member xs y then insert y (meet xs ys)
                   else meet xs ys

minus xs empty = xs
minus xs (insert y ys) = minus (delete y xs) ys
```

Mulțimi - reprezentarea cu liste

- Presupunând că a este instanță a clasei Eq, mulțimile se pot reprezenta cu liste
- Funcția abstr:

```
abstr :: [a] -> Set a
abstr = foldr insert empty
```

- Funcția de validare poate avea două variante:

```
valid xs = True
```

  - În acest caz orice listă reprezintă o mulțime. Operațiile se implementează uşor dar sunt dezavantaje majore

```
valid xs = nonduplicated xs
```

  - Se obțin rezultate mai bune la inserție şi reuniune

Mulțimi - reprezentarea cu liste

- Dacă orice listă este o reprezentare validă atunci:

```
member xs x = some(==x) xs
insert x xs = x:xs
delete x xs = filter(\= x) xs
union xs ys = xs ++ ys
minus xs ys = filter(not.member ys) xs

some :: (a -> Bool) -> [a] -> Bool
some p = or.map p
```

- Avantaj: insert necesită timp constant
- Dezavantaj: lista poate fi mult mai mare decât mulțimea. Dacă n este lungimea reprezentării (listei), delete este de complexitate $\Theta(n)$ iar minus $\Theta(n^2)$, în timp ce mulțimea poate să aibă m elemente cu m << n.

Mulțimi – reprezentarea cu liste

- Dacă se restricționează reprezentarea la liste cu elemente distincte atunci, o primă variantă de implementare:

```
insert x xs = x:filter(\=x)xs
union xs ys = xs ++ filter(not.member xs)ys
```

- Dar complexitatea este mare: $\Theta(n)$ respectiv $\Theta(n^2)$.

- Dacă presupunem a instanță a lui Ord şi impunem pentru validare liste strict ordonate atunci:

```
member xs x = if null xs then False
                  else(x == head ys)
        where ys = dropWhhile (<x) xs
union [] ys = ys
union (x:xs)(y:ys) | (x<y)  = x:union xs (y:ys)
                   | (x==y) = x:union xs ys
                   | (x>y)  = y:union (x:xs)ys
```

Mulțimi - reprezentarea cu arbori

```
module Set(Set, empty, isEmpty, member,insert,delete) where

data Set a = Null | Fork(Set a) a (Set a)
    deriving (Show)

empty :: Set a
empty = Null

isEmpty :: Set a -> Bool
isEmpty Null = True
isEmpty(Fork xt y zt) = False

member :: (Ord a) => Set a -> a -> Bool
member Null x = False
member (Fork xt y zt) x
    | (x < y)  = member xt x
    | (x == y) = True
    | (x > y)  = member zt x
```

Mulțimi - reprezentarea cu arbori

```
insert :: (Ord a) => a -> Set a -> Set a
insert x Null = Fork Null x Null
insert x (Fork xt y zt)
     | (x < y)  = Fork(insert x xt) y zt
     | (x == y) = Fork xt y zt
     | (x > y)  = Fork xt y (insert x zt)

delete :: (Ord a) => a -> Set a -> Set a
delete x Null =  Null
delete x (Fork xt y zt)
     | (x < y)  = Fork(delete x xt) y zt
     | (x == y) = join xt  zt
     | (x > y)  = Fork xt y (delete x zt)

join :: Set a -> Set a -> Set a
join xt yt = if isEmpty yt then xt else Fork xt y zt
          where (y, zt) = splitTree yt

splitTree :: Set a -> (a, Set a)
splitTree(Fork xt y zt) = if isEmpty xt then (y, zt) else (u, Fork vt y zt)
                    where(u, vt) = splitTree xt
```

Mulțimi – reprezentarea cu arbori

- Probleme la reprezentarea cu arbori:
  - timpul de execuție pentru member, insert delete depind de înălțimea arborelui
  - o mulțime cu n elemente poate fi reprezentată cu un arbore de înălțime log(n+1)
  - pentru păstrarea relației h = O(log n) - arbori echilibrați (arbori AVL)