1. Funcția `map`: Aplică o funcție dată fiecărui element al unei liste și returnează rezultatele într-o nouă listă.

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

2. Funcția `filter`: Selectează elementele dintr-o listă care satisfac o anumită condiție și le returnează într-o nouă listă.

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ []                 = []
filter p (x:xs) | p x       = x : filter p xs
                | otherwise = filter p xs
```

3. Funcția `foldl` (left fold): Aplică o funcție binară asupra elementelor unei liste, folosind un acumulator și parcurgerea listei de la stânga la dreapta.

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ acc []     = acc
foldl f acc (x:xs) = foldl f (f acc x) xs
```

4. Funcția `foldr` (right fold): Aplică o funcție binară asupra elementelor unei liste, folosind un acumulator și parcurgerea listei de la dreapta la stânga.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ acc []     = acc
foldr f acc (x:xs) = f x (foldr f acc xs)
```

5. Funcția `zip`: Preia două liste și le combină într-o singură listă de perechi, conținând elementele corespondente.

```
zip :: [a] -> [b] -> [(a, b)]
zip [] _          = []
zip _ []          = []
zip (x:xs) (y:ys) = (x, y) : zip xs ys
```

6. Funcția `concat`: Concatenează o listă de liste într-o singură listă.

```
concat :: [[a]] -> [a]
concat []       = []
```

```
concat (x:xs)   = x ++ concat xs
```

7. Funcția `length`: Returnează lungimea unei liste.

```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

8. Funcția `reverse`: Inversează ordinea elementelor unei liste.

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

9. Funcția `take`: Preia primele n elemente dintr-o listă.

```
take :: Int -> [a] -> [a]
take _ []     = []
take n (x:xs)
  | n <= 0    = []
  | otherwise = x : take (n-1) xs
```

10. Funcția `drop`: Elimină primele n elemente dintr-o listă.

```
drop :: Int -> [a] -> [a]
drop _ []     = []
drop n lst@(x:xs)
  | n <= 0    = lst
  | otherwise = drop (n-1) xs
```

11. Funcția `maximum`: Returnează valoarea maximă dintr-o listă.

```
maximum :: (Ord a) => [a] -> a
maximum []     = error "Lista este goală."
maximum [x]    = x
maximum (x:xs) = max x (maximum xs)
```

12. Funcția `minimum`: Returnează valoarea minimă dintr-o listă.

```
minimum :: (Ord a) => [a] -> a
minimum []     = error "Lista este goală."
```

```
minimum [x]    = x
minimum (x:xs) = min x (minimum xs)
```

13. Funcția `elem`: Verifică dacă un element există într-o listă.

```
elem :: (Eq a) => a -> [a] -> Bool
elem _ []     = False
elem x (y:ys) = x == y || elem x ys
```

14. Funcția `notElem`: Verifică dacă un element nu există într-o listă.

```
notElem :: (Eq a) => a -> [a] -> Bool
notElem _ []     = True
notElem x (y:ys) = x /= y && notElem x ys
```

15. Funcția `null`: Verifică dacă o listă este goală.

```
null :: [a] -> Bool
null []    = True
null (_:_) = False
```

16. Funcția `head`: Returnează primul element dintr-o listă.

```
head :: [a] -> a
head []    = error "Lista este goală."
head (x:_) = x
```

17. Funcția `tail`: Elimină primul element dintr-o listă și returnează restul.

```
tail :: [a] -> [a]
tail []     = error "Lista este goală."
tail (_:xs) = xs
```

18. Funcția `init`: Elimină ultimul element dintr-o listă și returnează restul.

```
init :: [a] -> [a]
init []     = error "Lista
 este goală."
init [_]    = []
init (x:xs) = x : init xs
```

19. Funcția `last`: Returnează ultimul element dintr-o listă.

```
last :: [a] -> a
last []    = error "Lista este goală."
last [x]   = x
last (_:xs) = last xs
```

20. Funcția `replicate`: Creează o listă cu n copii ale unui element dat.

```
replicate :: Int -> a -> [a]
replicate n x
  | n <= 0    = []
  | otherwise = x : replicate (n-1) x
```

21. Funcția `zipWith`: Preia două liste și aplică o funcție binară asupra elementelor corespondente.

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith _ [] _          = []
zipWith _ _ []          = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

22. Funcția `any`: Verifică dacă orice element dintr-o listă satisface o anumită condiție.

```
any :: (a -> Bool) -> [a] -> Bool
any _ []     = False
any p (x:xs) = p x || any p xs
```

23. Funcția `all`: Verifică dacă toate elementele dintr-o listă satisfac o anumită condiție.

```
all :: (a -> Bool) -> [a] -> Bool
all _ []     = True
all p (x:xs) = p x && all p xs
```

24. Funcția `splitAt`: Desparte o listă în două la un anumit index.

```
splitAt :: Int -> [a] -> ([a], [a])
splitAt _ []     = ([], [])
splitAt n xs
  | n <= 0    = ([], xs)
  | otherwise = let (ys, zs) = splitAt (n-1) (tail xs) in (head xs : ys, zs)
```

25. Funcția `concatMap`: Map-ează o funcție asupra elementelor unei liste și apoi concatenează rezultatele.

```
concatMap :: (a -> [b]) -> [a] -> [b]
concatMap _ []     = []
concatMap f (x:xs) = f x ++ concatMap f xs
```

1. **Exercițiul:** Implementați o funcție `subsets :: [a] -> [[a]]` care primește o listă și returnează toate submulțimile acelei liste.

```
subsets :: [a] -> [[a]]
subsets [] = [[]]
subsets (x:xs) = subsets xs ++ map (x:) (subsets xs)
```

2. **Exercițiul:** Implementați o funcție `permutations :: [a] -> [[a]]` care primește o listă și returnează toate permutările posibile ale elementelor listei.

```
permutations :: [a] -> [[a]]
permutations [] = [[]]
permutations (x:xs) = concatMap (insertElem x) (permutations xs)
  where
    insertElem :: a -> [a] -> [[a]]
    insertElem x [] = [[x]]
    insertElem x (y:ys) = (x:y:ys) : map (y:) (insertElem x ys)
```

3. **Exercițiul:** Implementați o funcție `combinations :: Int -> [a] -> [[a]]` care primește un număr `k` și o listă și returnează toate combinațiile de `k` elemente posibile din lista dată.

```
combinations :: Int -> [a] -> [[a]]
combinations 0 _ = [[]]
combinations k xs = [x:ys | x:xs' <- tails xs, ys <- combinations (k-1) xs']
```