CSE130 - WI19

Final Review

Agenda

- PA-06 Tips
- Practice Final Insights

PA-06 Tips

PA06- All in one slide

- No predicate has to be larger than 3 or 4 sub-statements long
- Start by writing base cases
  - For zip, for example, consider empty lists
- For union, consider reusing previously written predicates
- **Bagof** and **isin** are very useful for part 2

In prolog, you write predicates that must be true in order to make a larger query true. Please do not think of it in terms of '*how do I compute this value*'. This of it in terms of '*what has to be true in order for this query to return true*'

Semantics & Type-systems (and other formalisms)

This looks difficult but it's actually just *flashy*

4.1 Reduction 1 [10 points]

Complete the following reduction derivation, where

```
E = [f -> <[], \x y -> x + y>]
```

```
[______] ------------------------------------------------------

         E,          => E,

[______] ------------------------------------------------------

         E,          => E,

[______] ------------------------------------------------------

         E, f 1 2    => E,
```

Strategy: pattern-recognition

4.1 Reduction 1 [10 points]

Complete the following reduction derivation, where

```
E = [f -> <[], \x y -> x + y>]
```

```
[______] ------------------------------------------------------

         E,          => E,

[______] ------------------------------------------------------

         E,          => E,

[______] ------------------------------------------------------

         E, f 1 2    => E,
```

Step 1: Identify the right rules

Operational semantics:

```
[Var]      E, x => E, E[x]                 if x in dom(E)

[Add]      E, n1 + n2 => E, n              where n == n1 + n2

                E, e1 => E', e1'
[Add-L]   ---------------------------
          E, e1 + e2 => E', e1' + e2

                E, e2 => E', e2'
[Add-R]   ---------------------------
          E, n1 + e2 => E', n1 + e2'

[Let]      E, let x = v in e2 => E[x->v], e2

                          E, e1 => E', e1'
[Let-Def] ------------------------------------------------
          E, let x = e1 in e2 => E', let x = e1' in e2

[Abs]      E, \x -> e => E, <E,\x -> e>

[App]      E, <E1,\x -> e> v => E1[x->v], e

              E, e1 => E', e1'
[App-L]   ----------------------
          E, e1 e2 => E', e1' e2

            E, e => E', e'
[App-R]   ------------------
          E, v e => E', v e'
```

V.S

Typing rules:

```
[T-Num]   G |- n :: Int

                G |- e1 :: Int   G |- e2 :: Int
[T-Add]   --------------------------------
                    G |- e1 + e2 :: Int

[T-Var]   G |- x :: S                 if x:S in G

            G, x:T1 |- e :: T2
[T-Abs]   ------------------------
          G |- \x -> e :: T1 -> T2

                G |- e1 :: T1 -> T2   G |- e2 :: T1
[T-App]   ------------------------------------
                      G |- e1 e2 :: T2

                G |- e1 :: S   G, x:S |- e2 :: T
[T-Let]   ---------------------------------
             G |- let x = e1 in e2 :: T

                G |- e :: forall a . S
[T-Inst]  ----------------------
           G |- e :: [a / T] S

                 G |- e :: S
[T-Gen]   ---------------------- if not (a in FTV(G))
          G |- e :: forall a . S
```

Step 1: Identify the right rules

Operational semantics:

```
[Var]      E, x => E, E[x]                 if x in dom(E)

[Add]      E, n1 + n2 => E, n              where n == n1 + n2

                E, e1 => E', e1'
[Add-L]    ---------------------------
           E, e1 + e2 => E', e1' + e2

                E, e2 => E', e2'
[Add-R]    ---------------------------
           E, n1 + e2 => E', n1 + e2'

[Let]      E, let x = v in e2 => E[x->v], e2

                          E, e1 => E', e1'
[Let-Def]  ------------------------------------------
           E, let x = e1 in e2 => E', let x = e1' in e2

[Abs]      E, \x -> e => E, <E,\x -> e>

[App]      E, <E1,\x -> e> v => E1[x->v], e

              E, e1 => E', e1'
[App-L]    ----------------------
           E, e1 e2 => E', e1' e2

             E, e => E', e'
[App-R]    ------------------
           E, v e => E', v e'
```

V.S

Typing rules:

```
[T-Num]    G |- n :: Int

                G |- e1 :: Int   G |- e2 :: Int
[T-Add]    --------------------------------
                    G |- e1 + e2 :: Int

[T-Var]    G |- x :: S                if x:S in G

             G, x:T1 |- e :: T2
[T-Abs]    ------------------------
           G |- \x -> e :: T1 -> T2

                G |- e1 :: T1 -> T2   G |- e2 :: T1
[T-App]    ------------------------------------
                      G |- e1 e2 :: T2

                G |- e1 :: S   G, x:S |- e2 :: T
[T-Let]    ---------------------------------
              G |- let x = e1 in e2 :: T

                G |- e :: forall a . S
[T-Inst]   ----------------------
            G |- e :: [a / T] S

                   G |- e :: S
[T-Gen]    ---------------------- if not (a in FTV(G))
           G |- e :: forall a . S
```

Step 2: Remember your parenthesis (left assoc)

4.1 Reduction 1 [10 points]

Complete the following reduction derivation, where

```
E = [f -> <[], \x y -> x + y>]
```

```
[______] ------------------------------------------------------
         E,          => E,

[______] ------------------------------------------------------
         E,          => E,

[______] ------------------------------------------------------
         E, f 1 2    => E,
```

Step 2: Remember your parenthesis (left assoc)

a b c d e

Step 2: Remember your parenthesis (left assoc)

(a b) c d e

Step 2: Remember your parenthesis (left assoc)

((a b) c) d e

Step 2: Remember your parenthesis (left assoc)

(((a b) c) d)e

Step 2: Remember your parenthesis (left assoc)

4.1 Reduction 1 [10 points]

Complete the following reduction derivation, where

```
E = [f -> <[], \x y -> x + y>]
```

```
[______] ------------------------------------------------------
          E,          => E,

[______] ------------------------------------------------------
          E,          => E,

[______] ------------------------------------------------------
          E, (f 1) 2  => E,
```

Step 3: Pattern matching (+ intuition)

Step 3: Pattern matching (+ intuition)

```
[______] ------------------------------------------------------
```

```
E, [f 1] 2     => E,
```

Operational semantics:

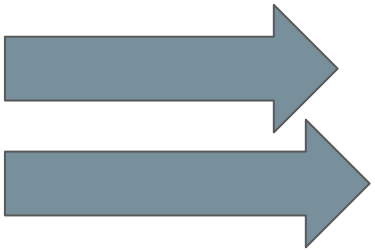
```
[Var]      E, x => E, E[x]                    if x in dom(E)

[Add]      E, n1 + n2 => E, n                 where n == n1 + n2

                E, e1 => E', e1'
[Add-L]    ---------------------------
           E, e1 + e2 => E', e1' + e2

                E, e2 => E', e2'
[Add-R]    ---------------------------
           E, n1 + e2 => E', n1 + e2'

[Let]      E, let x = v in e2 => E[x->v], e2

                          E, e1 => E', e1'
[Let-Def]  ---------------------------------------------
           E, let x = e1 in e2 => E', let x = e1' in e2

[Abs]      E, \x -> e => E, <E,\x -> e>

[App]      E, <E1,\x -> e> v => E1[x->v], e

              E, e1 => E', e1'
[App-L]    ----------------------
           E, e1 e2 => E', e1' e2

             E, e => E', e'
[App-R]    ------------------
           E, v e => E', v e'
```

Step 3: Pattern matching (+ intuition)

```
[______] ------------------------------------------------------
```

```
E, f 1 2    => E,
```

It’s gotta be some sort of application



Operational semantics:

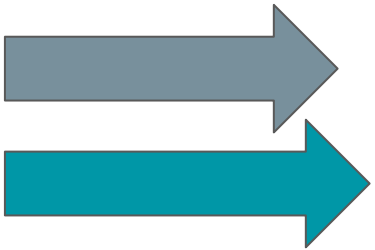
```
[Var]      E, x => E, E[x]                    if x in dom(E)

[Add]      E, n1 + n2 => E, n                 where n == n1 + n2

                E, e1 => E', e1'
[Add-L]    ---------------------------
           E, e1 + e2 => E', e1' + e2

                E, e2 => E', e2'
[Add-R]    ---------------------------
           E, n1 + e2 => E', n1 + e2'

[Let]      E, let x = v in e2 => E[x->v], e2

                          E, e1 => E', e1'
[Let-Def]  ------------------------------------------------
           E, let x = e1 in e2 => E', let x = e1' in e2

[Abs]      E, \x -> e => E, <E,\x -> e>

[App]      E, <E1,\x -> e> v => E1[x->v], e

              E, e1 => E', e1'
[App-L]    ----------------------
           E, e1 e2 => E', e1' e2

             E, e => E', e'
[App-R]    ------------------
           E, v e => E', v e'
```

Step 3: Pattern matching (+ intuition)

```
[_____] ------------------------------------------------------
```

```
E, |f 1| 2    => E,
```

Since we have multiple args & f is not in “lambda form”



Operational semantics:

```
[Var]      E, x => E, E[x]                    if x in dom(E)

[Add]      E, n1 + n2 => E, n                 where n == n1 + n2

                E, e1 => E', e1'
[Add-L]    ------------------------
           E, e1 + e2 => E', e1' + e2

                E, e2 => E', e2'
[Add-R]    ------------------------
           E, n1 + e2 => E', n1 + e2'

[Let]      E, let x = v in e2 => E[x->v], e2

                          E, e1 => E', e1'
[Let-Def]  ---------------------------------------------
           E, let x = e1 in e2 => E', let x = e1' in e2

[Abs]      E, \x -> e => E, <E,\x -> e>

[App]      E, <E1,\x -> e> v => E1[x->v], e

              E, e1 => E', e1'
[App-L]    ----------------------
           E, e1 e2 => E', e1' e2

             E, e => E', e'
[App-R]    ------------------
           E, v e => E', v e'
```

Step 4: Rinse and repeat

4.1 Reduction 1 [10 points]

Complete the following reduction derivation,

```
E = [f -> <[], \x y -> x + y>]
```

```
[_____] ----------------------------
        E,          => E,

[_____] ----------------------------
        E,          => E,

[_____] ----------------------------
        E, f 1 2    => E,
```

$$E = [f \to \langle [], \backslash x\ y \to x + y \rangle]$$

$$\text{[App-L]}\ \frac{E, f\ 1 \Rightarrow E, \langle [], \backslash x\ y \to x + y \rangle\ 1}{E, f\ 1\ 2 \Rightarrow E, \langle [], \backslash x\ y \to x + y \rangle\ 1\ 2}$$

Step 4: Rinse and repeat

4.1 Reduction 1 [10 points]

Complete the following reduction derivation,

```
E = [f -> <[], \x y -> x + y>
```

```
[_____] ----------------------------
        E,          => E,

[_____] ----------------------------
        E,          => E,

[_____] ----------------------------
        E, f 1 2    => E,
```

```
E = [f -> <[], \x y -> x + y>

        E, f => E, <[], \x y -> x + y>
[App-L] ------------------------------
        E, f 1  =>  E, <[], \x y -> x + y> 1
[App-L] ------------------------------------
        E, f 1 2  =>  E, <[], \x y -> x + y> 1 2
```

Step 4: Rinse and repeat

4.1 Reduction 1 [10 points]

Complete the following reduction derivation,

```
E = [f -> <[], \x y -> x + y>]
```

```
[_____] ----------------------------
        E,          => E,

[_____] ----------------------------
        E,          => E,

[_____] ----------------------------
        E, f 1 2    => E,
```

```
E = [f -> <[], \x y -> x + y>]

[Var]     ----------------------
          E, f => E, <[], \x y -> x + y>

[App-L]   ------------------------------
          E, f 1  =>  E, <[], \x y -> x + y> 1

[App-L]   ----------------------------------
          E, f 1 2  =>  E, <[], \x y -> x + y> 1 2
```

In case you didn't know

When rules / judgements at the top of the problem (root of the tree) are probably going to be those with no premises / no dotted lines.

Operational semantics:

```
[Var]    E, x => E, E[x]          if x in dom(E)

[Add]    E, n1 + n2 => E, n       where n == n1 + n2

[Abs]    E, \x -> e => E, <E,\x -> e>

[App]    E, <E1,\x -> e> v => E1[x->v], e
```

In case you didn't know (pt.2)

If you see multiple branches, then that should limit the number of valid rules!

4.3 Typing 1 [10 points]

Complete the following typing derivation

```
[______]----------------------        ----------------------[______]


[______]-------------------------------------------------


[______]-------------------------------------------------

        [] |- \x -> x + 5 ::
```

Typing rules:

```
          G |- e1 :: Int   G |- e2 :: Int
[T-Add]   -------------------------------
                G |- e1 + e2 :: Int
```

```
          G |- e1 :: T1 -> T2   G |- e2 :: T1
[T-App]   ------------------------------------
                    G |- e1 e2 :: T2
```

```
          G |- e1 :: S   G, x:S |- e2 :: T
[T-Let]   --------------------------------
             G |- let x = e1 in e2 :: T
```

Prolog

Hack: There's only so many question we can ask

Think about:

- Deleting from a list
- Finding duplicates in a list
- Finding the index of an element
- Doing merge sort, selection sort, bubble sort
- Finding the median in a list
- Sorted insertion in a sorted array
- Find the k-smallest element of a list

(I do not necessarily know how to do all this nor if these are all possible with your current toolset, but these are the kinds of questions I'd ponder on)

Please please please: have the right mindset

Prolog is **not** about defining *how* to do calculate a value

How NOT to write prolog

Prolog is **not** about defining *how* to calculate a value
Prolog is **not** about defining *how* to calculate a value
Prolog is **not** about defining *how* to calculate a value
Prolog is **not** about defining *how* to calculate a value
Prolog is **not** about defining *how* to calculate a value
Prolog is **not** about defining *how* to calculate a value
Prolog is **not** about defining *how* to calculate a value
Prolog is **not** about defining *how* to calculate a value
Prolog is **not** about defining *how* to calculate a value
Prolog is **not** about defining *how* to calculate a value

How to write prolog

Prolog is about **stating what has to be true about the result and letting prolog find that result for you!**

For example

5.3 Selection Sort [10 points]

```
selection_sort([],[]).
selection_sort([X|Xs],[Y|Ys]) :- list_min(X, Xs, Y)
                               , insert(Y,Zs,[X|Xs])
                               , selection_sort(Zs,Ys).
```

For example

5.3 Selection Sort [10 points]

This is a “true statement”

```
selection_sort([],[]).
selection_sort([X|Xs],[Y|Ys]) :- list_min(X, Xs, Y)
                               , insert(Y,Zs,[X|Xs])
                               , selection_sort(Zs,Ys).
```

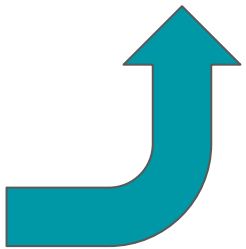
For example

5.3 Selection Sort [10 points]

```
selection_sort([],[]).
selection_sort([X|Xs],[Y|Ys]) :- list_min(X, Xs, Y)
                               , insert(Y,Zs,[X|Xs])
                               , selection_sort(Zs,Ys).
```

This is a “true statement”

These three have to be true in order for `selection_sort([X|Xs, [Y|Ys])` to be true.



A common mistake: mixing syntax

In previous exams, we've seen students answer the prolog question with:

- Pattern-matching
- Lambda expressions
- Haskell list utilities

Please remember that this is not valid Prolog syntax. Do be familiar with things like

- Bagof
- append
- reverse
- ‘ ’
- _
- [X|rest]
- ‘is’

HOF

List of Higher Order Functions

- Filter
- Map
- Fold(left/right)

Filter

- Used to filter a list.
- Takes a function as parameter which outputs a boolean for each element of a list.
- It is generally used when you are provided with a list and you need to remove certain elements from a list based on some function.
- Remove the elements which are <0 from the given list.

Map

- Takes a function as parameter and applies that function to each element of the list individually.
- It is generally used when you need to modify the elements of the given list to the output of a certain function.
- Example: Add 2 to each element of the list.

Foldl/Foldr

- Fold takes a function and a base value as parameters and applies that function over all the elements of the list together.
- It is generally used when you need to output a value which is dependent on all the elements of the list.
- Example : Sum of elements of list, Product of all elements of the list
- Remember the difference between Foldl and Foldr

Foldl/Foldlr

```
foldr (+) 0 [1, 2, 3, 4]
  ==> 1 + (foldr (+) 1 [2, 3, 4])
  ==> 1 + (2 + (foldr (+) 0 [3, 4]))
  ==> 1 + (2 + (3 + (foldr (+) 0 [4])))
  ==> 1 + (2 + (3 + (4 + (foldr (+) 0 []))))
  ==> 1 + (2 + (3 + (4 + 0)))
```

```
foldl (+) 0                    [1, 2, 3, 4]
  ==> helper 0                 [1, 2, 3, 4]
  ==> helper (0 + 1)              [2, 3, 4]
  ==> helper ((0 + 1) + 2)           [3, 4]
  ==> helper (((0 + 1) + 2) + 3)        [4]
  ==> helper ((((0 + 1) + 2) + 3) + 4)  []
  ==> ((((0 + 1) + 2) + 3) + 4)
```

3.1 List Reversal

3.1 List reversal [5 pts]

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

3.1 List Reversal

3.1 List reversal [5 pts]

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

reverse xs=

3.1 List Reversal

3.1 List reversal [5 pts]

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

reverse xs= foldl

3.1 List Reversal

3.1 List reversal [5 pts]

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

reverse xs= foldl (\acc x -> x:acc)

3.1 List Reversal

3.1 List reversal [5 pts]

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

```
reverse xs= foldl (\acc x -> x:acc) []
```

3.1 List Reversal

3.1 List reversal [5 pts]

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

```
reverse xs= foldl (\acc x -> x:acc) [] xs
```

3.2 Absolute Values

3.2 Absolute values [10 pts]

```
absValues :: [Int] -> [Int]
absValues []     = []
absValues (x:xs)
  | x < 0       = (-x):(absValues xs)
  | otherwise   =   x :(absValues xs)
```

absValues xs =

3.2 Absolute Values

3.2 Absolute values [10 pts]

```
absValues :: [Int] -> [Int]
absValues []     = []
absValues (x:xs)
  | x < 0       = (-x):(absValues xs)
  | otherwise   =   x :(absValues xs)
```

absValues xs = map

3.2 Absolute Values

3.2 Absolute values [10 pts]

```
absValues :: [Int] -> [Int]
absValues []     = []
absValues (x:xs)
  | x < 0       = (-x):(absValues xs)
  | otherwise   =   x :(absValues xs)
```

absValues xs = map (\x->abs x) xs

3.2 Absolute Values

3.2 Absolute values [10 pts]

```
absValues :: [Int] -> [Int]
absValues []     = []
absValues (x:xs)
  | x < 0       = (-x):(absValues xs)
  | otherwise   =   x :(absValues xs)
```

absValues xs = map (\x->abs x) xs

abs x = if x<0 then (-x) else x

3.3 Remove duplicates

```haskell
dedup :: [Int] -> [Int]
dedup [] = []
dedup (x:xs) = x:(remove x (dedup xs))
  where
    remove x [] = []
    remove x (y:ys)
      | x == y    =    remove x ys
      | otherwise = y:(remove x ys)
```

dedup xs = ??

3.3 Remove duplicates

```haskell
dedup :: [Int] -> [Int]
dedup [] = []
dedup (x:xs) = x:(remove x (dedup xs))
  where
    remove x [] = []
    remove x (y:ys)
      | x == y    =    remove x ys
      | otherwise = y:(remove x ys)
```

dedup xs = foldr f [] xs

3.3 Remove duplicates

```haskell
dedup :: [Int] -> [Int]
dedup [] = []
dedup (x:xs) = x:(remove x (dedup xs))
  where
    remove x [] = []
    remove x (y:ys)
      | x == y    =    remove x ys
      | otherwise = y:(remove x ys)
```

```haskell
dedup xs = foldr f [] xs

  where

    f x ys = ??
```

3.3 Remove duplicates

```haskell
dedup :: [Int] -> [Int]
dedup [] = []
dedup (x:xs) = x:(remove x (dedup xs))
  where
    remove x [] = []
    remove x (y:ys)
      | x == y    =    remove x ys
      | otherwise = y:(remove x ys)
```

```haskell
dedup xs = foldr f [] xs
  where
    f x ys = x : (filter (/= x) ys)
```

3.4 Insertion Sort

3.4 Insertion Sort* [20 pts]

```
sort :: [Int] -> [Int]
sort [] = []
sort (x:xs) = insert x (sort xs)
  where
    insert x []     = [x]
    insert x (y:ys) = if x <= y
                      then x:y:ys
                      else y:(insert x ys)
```

sort xs = ??

3.4 Insertion Sort

3.4 Insertion Sort* [20 pts]

```
sort :: [Int] -> [Int]
sort [] = []
sort (x:xs) = insert x (sort xs)
  where
    insert x []     = [x]
    insert x (y:ys) = if x <= y
                      then x:y:ys
                      else y:(insert x ys)
```

sort xs = foldl ??

3.4 Insertion Sort

3.4 Insertion Sort* [20 pts]

```
sort :: [Int] -> [Int]
sort [] = []
sort (x:xs) = insert x (sort xs)
  where
    insert x []     = [x]
    insert x (y:ys) = if x <= y
                      then x:y:ys
                      else y:(insert x ys)
```

```
sort xs = foldl insert [] xs

  where

    Insert ys x = ??
```

3.4 Insertion Sort

3.4 Insertion Sort* [20 pts]

```
sort :: [Int] -> [Int]
sort [] = []
sort (x:xs) = insert x (sort xs)
  where
    insert x []     = [x]
    insert x (y:ys) = if x <= y
                      then x:y:ys
                      else y:(insert x ys)
```

```
sort xs = foldl insert [] xs
  where
    Insert ys a = append (filter (< x) ys)
                         (x: filter (>= x) ys)

    append=??
```

3.4 Insertion Sort

3.4 Insertion Sort* [20 pts]

```
sort :: [Int] -> [Int]
sort [] = []
sort (x:xs) = insert x (sort xs)
  where
    insert x []     = [x]
    insert x (y:ys) = if x <= y
                      then x:y:ys
                      else y:(insert x ys)
```

```
sort xs = foldl insert [] xs
  where
    Insert ys x = append (filter (< x) ys)
                         (x: filter (>= x) ys)

    append xs ys = foldr (:) ys xs
```

Datatypes and Recursion

- Remember the correct format of pattern matching

```
data Entry =
    File String Int       -- file: name and size
  | Dir String [Entry]    -- directory: name and child entries
```

Datatypes and Recursion

```
size :: Entry -> Int
size (File _ s) = s
size (Dir _ fs)      = dirSize fs
  where
    dirSize []     = 0
    dirSize (f:fs) = size f + dirSize fs
```

Datatypes and Recursion

```
Size :: Entry -> Int

Size x = case x of

        (File _ s) -> s

        (Dir _ fs) -> dirSize fs

          Where

                dirSize :: [Entry] -> Int

                dirSize [] = 0

                dirSize (f:fs) = size f + dirSize fs
```

How not to pattern match

```
Size :: Entry -> Int

Size x

    | x == (File _ s) = s

...
```

How not to pattern match

Size :: Entry -> Int

Size x = if x == (File _ s) then s ...

Datatypes and Recursion

- Make sure you add the base case for the recursion
- It is recommended to see how your program will run on a few small test cases (specially all the given test cases)
- It is easier to write non-tail recursive functions. You will be given full points for a correct recursive function (unless the question explicitly states tail-recursive program)

Datatypes and Recursion

- Make sure you carefully look at the types of the recursive function, specially when it involves pattern matching.

Datatypes and Recursion

- Make sure you carefully look at the types of the recursive function, specially when it involves pattern matching.

  The following program is WRONG:

  ```
  Size :: Entry -> Int

  Size (File _ s) = s

  Size (Dir _ fs) = Size fs
  ```

Datatypes and Recursion

- Make sure you carefully look at the types of the recursive function, specially when it involves pattern matching.
  The following program is also WRONG:

```
Size :: Entry -> Int

Size (File _ s) = s

Size (Dir _ fs) = dirSize fs

    Where

        dirSize [] = 0

        dirSize (x:xs) = size x + size xs
```

Datatypes and Recursion

- Make sure you carefully look at the types of the recursive function, specially when it involves pattern matching.
- It would help if you write type for helpers and match the types when you are doing the recursive step.