Programming 2
Tutorial 4

Imperative Programming

Nedim Šišić

2023

Exercises

- Use both fold functions separately to apply logical implication to a given list. Apply both functions on the list [true;true;true] and an initial value of false.

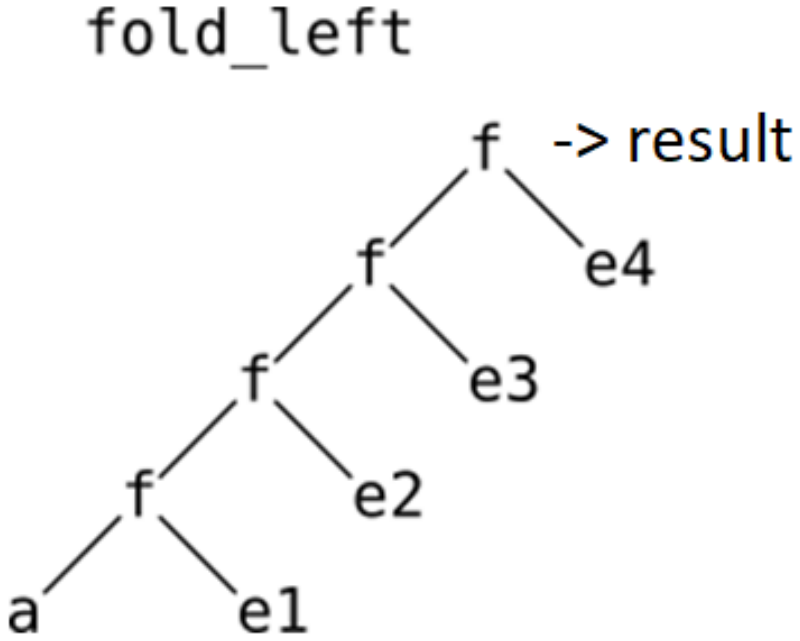
Exercises

- Use both fold functions separately to apply logical implication to a given list. Apply both functions on the list [true;true;true] and an initial value of false.

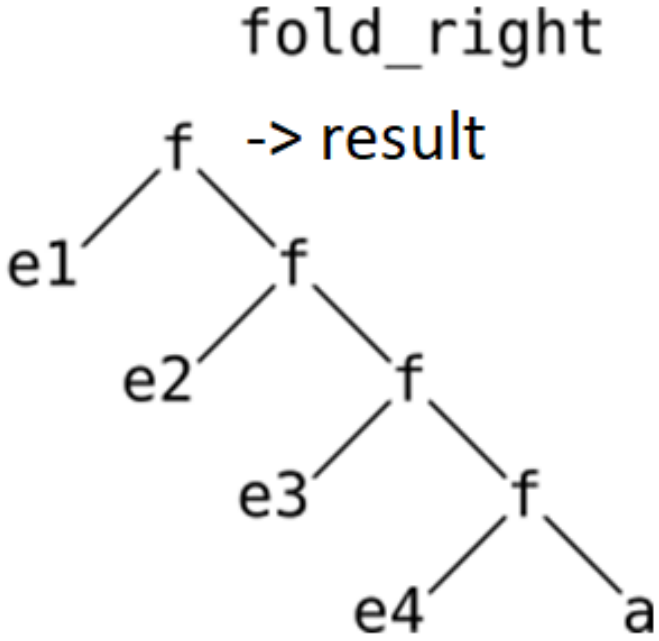
```
let implies a b =
  match (a,b) with
    | (true,false) -> false;
    | _ -> true;;

List.fold_left implies false [true; true; true];;

List.fold_right implies [true; true; true] false;;
```

Lists – folding left and folding right

List.fold_left f a [e1; e2; e3; e4];;



List.fold_right f [e1; e2; e3; e4] a;;



Exercises

- Declare local variables a, b, and c and return a*b + a*c - b*c
- Declare a global function *powerOf8*, that applies a local function *square* to an integer parameter three times

Exercises

- Declare local variables a, b, and c and return a*b + a*c - b*c

```
let a = 5 in let b = 2 in let c = 4 in a*b + a*c - b*c;;
```

- Declare a global function *powerOf8*, that applies a local function *square* to an integer parameter three times

```
let powerOf8 x = let square x = x*x in square (square ( square (x)));;
```

Exercises

We have the following variables:

```
let a = ref 10;;
let b = ref 10;;
let c = a;;
```

- Test each pair of variables for *structural equality* ("=")
- Test each pair of variables for *physical equality* ("==")
- See if *assigning* (“ := “) a new value to *a* changes *b* and *c*

Exercises

- Test each pair of variables for *structural equality* ("=")

```
a = b;; a = c;; b = c;;
```

- Test each pair of variables for *physical equality* ("==")

```
a == b;; a == c;; b == c;;
```

- See if *assigning* (:=) a new value to *a* changes *b* and *c*

```
a := 5;;
!b;; !c;;
```

Exercises

- Recursive functions without using the keyword rec – factorial:

```
let refFun = ref (fun x -> x);;

let fact n =
  match n with
  | 0 -> 1
  | _ -> n * !refFun (n-1);;

refFun := fact;;

fact 5;;
```

Exercises

- Write a non-recursive function *sumToN* using function referencing, equivalent to its recursive version:

```
let rec sumToN n =
  match n with
  | 0 -> 0
  | _ -> n + sumToN (n - 1);;
```

Exercises

- Write a non-recursive function *sumToN* using function referencing, equivalent to its recursive version:

```
let refFun = ref (fun x -> x);;

let sumToN_ n =
   match n with
  | 0 -> 0
  | _ -> n + !refFun (n-1);;

refFun := sumToN_;;
```

Exercises

Define a variable counter initialized to 0:

```
let counter = ref 0;; .
```

Write a *next_val* function, with no parameters*, which in each call

increases the value of counter by 1 and **returns** the new value.

(*you can declare the function like "let next_val _ = ..." or "let next_val () = ...")

Exercises

Define a variable counter initialized to 0:

```
let counter = ref 0;; .
```

Write a *next_val* function, with no parameters*, which in each call

increases the value of counter by 1 and **returns** the new value.

(*you can declare the function like "let next_val _ = ..." or "let next_val () = ...")

```
let counter = ref 0;;
let next_val _ =
  counter := (!counter) + 1;
  !counter;;
```

Exercises

Write a *next_val2* function, with no parameters, which in each call **increases** the value of a variable *counter2* by 1 and **returns** the new value, with *counter2* **accessible** only inside the scope of *next_val*.

Exercises

Write a *next_val2* function, with no parameters, which in each call **increases** the value of a variable *counter2* by 1 and **returns** the new value, with *counter2* **accessible** only inside the scope of *next_val*.

```
let next_val_2 =
  let counter_2 = ref 0 in
    fun () ->
      incr counter_2;
      !counter_2;;
```