# Functional Programming

## Higher-Order Functions

H. Turgut Uyar

2013-2017



## License





## Topics





## First Class Values

- first class values can be:
- assigned
- composed with other values
- passed as parameters
- returned as function results
- in functional programming, functions are first class values

## Function Order

- first order functions
- only accept data as parameter, and
- only return data as result

- higher-order functions
- take functions as parameters, or
- return functions as result



## First Order Function Examples

sum up the squares in a range

```
-- sqr :: Integer -> Integer
sumSqr :: Integer -> Integer -> Integer
sumSqr a b
  | a > b     = 0
  | otherwise = sqr a + sumSqr (a + 1) b
```

sum up the factorials in a range

```
-- fac :: Integer -> Integer
sumFac :: Integer -> Integer -> Integer
sumFac a b
  | a > b     = 0
  | otherwise = fac a + sumFac (a + 1) b
```



## Higher-Order Function Example

- note the pattern

```
sumFun a b
  | a > b     = 0
  | otherwise = fun a + sumFun (a + 1) b
```

- send the function as parameter

```
sumF f a b
  | a > b     = 0
  | otherwise = f a + sumF f (a + 1) b

sumSqr a b = sumF sqr a b
sumFac a b = sumF fac a b
```



## Higher-Order Function Example

- what is the type of `f`?

```
Integer -> Integer
```

- what is the type of `sumF`?

```
(Integer -> Integer) -> Integer -> Integer -> Integer
```

## Higher-Order Function Example

### Python

```python
def sum_f(f, a, b):
    total = 0
    while a <= b:
        total += f(a)
        a += 1
    return total

def sqr(x):
    return x * x

def sum_sqr(a, b):
    return sum_f(sqr, a, b)
```



## Higher-Order Function Example

### C

```c
int sum_f(int (*f)(int), int a, int b)
{
    int total = 0;
    while (a <= b) {
        total += f(a);
        a += 1;
    }
    return total;
}
```



## Higher-Order Function Example

### C

```c
int sqr(int x)
{
    return x * x;
}

int sum_sqr(int a, int b)
{
    return sum_f(sqr, a, b);
}
```



## Higher-Order Function Example

### Rock - Paper - Scissors

- parameterize `generateMatch` regarding both strategies

```haskell
type Strategy = [Move] -> Move

generateMatch :: Strategy -> Strategy -> Integer
                 -> Match
generateMatch _  _  0 = ([], [])
generateMatch sA sB n = step (generateMatch sA sB (n - 1))
  where
    step :: Match -> Match
    step (movesA, movesB) = (sA movesB : movesA,
                             sB movesA : movesB)
```

## Example: Sorting

insertion sort

```
ins :: Integer -> [Integer] -> [Integer]
ins n []          = [n]
ins n xs@(x':xs')
  | n <= x'       = n : xs
  | otherwise     = x' : ins n xs'

iSort :: [Integer] -> [Integer]
iSort []     = []
iSort (x:xs) = ins x (iSort xs)
```



## Example: Sorting

parameterize `iSort` regarding precedes function

```
ins' :: (Integer -> Integer -> Bool)
          -> Integer -> [Integer] -> [Integer]
ins' p n []          = [n]
ins' p n xs@(x':xs')
  | p n x'           = n : xs
  | otherwise        = x' : ins' p n xs'

iSort' :: (Integer -> Integer -> Bool)
            -> [Integer] -> [Integer]
iSort' p []     = []
iSort' p (x:xs) = ins' p x (iSort' p xs)

-- iSort' (<=) [4, 5, 3]       ~> [3, 4, 5]
-- iSort' (>)  [4, 5, 3]       ~> [5, 4, 3]
```



## Example: Sorting

```
ins' :: (a -> a -> Bool) -> a -> [a] -> [a]
ins' p n []          = [n]
ins' p n xs@(x':xs')
  | p n x'           = n : xs
  | otherwise        = x' : ins' p n xs'

iSort' :: (a -> a -> Bool) -> [a] -> [a]
iSort' p []     = []
iSort' p (x:xs) = ins' p x (iSort' p xs)

-- iSort' (<=) [4, 5, 3]       ~> [3, 4, 5]
-- iSort' (<=) ["b", "a", "c"] ~> ["a", "b", "c"]
```



## Example: Sorting

- in C, `qsort` takes comparison function as parameter

```
typedef struct {
    int num, denom;
} rational;

rational items[] = {{3, 2}, {1, 3}, {2, 1}};
qsort(items, 3, sizeof(rational), compare_rationals);
```

## Sorting

```
int compare_rationals(const void *r1, const void *r2)
{
    const rational *x = r1, *y = r2;
    int diff = x->num * y->denom - y->num * x->denom;

    if (diff < 0)
        return -1;
    else if (diff > 0)
        return 1;
    else
        return 0;
}
```



## Sorting

- in Python, `sorted` takes key function as parameter

```
def second(p):
    return p[1]

def value(p):
    return p[0] / p[1]

items = [(3, 2), (1, 3), (2, 1)]

# sorted(items)             ~> [(1, 3), (2, 1), (3, 2)]
# sorted(items, key=second) ~> [(2, 1), (3, 2), (1, 3)]
# sorted(items, key=value)  ~> [(1, 3), (3, 2), (2, 1)]
```



## Anonymous Functions

- no need to name small functions that are not used anywhere else
  → anonymous functions

```
\x1 x2 ... -> e
```

- `f x = e` : `f = \x -> e`

example

```
sumSqr :: Integer -> Integer -> Integer
sumSqr a b = sumF (\x -> x * x) a b
```



## Anonymous Functions

Python

```
lambda x1, x2, ...: e
```

examples

```
def sum_sqr(a, b):
    sum_func(lambda x: x * x, a, b)


sorted(items, key=lambda p: p[0] / p[1])
```

## Fixed Points

- $x$ is a *fixed point* of $f$:
  $f(x) = x$
- repeatedly apply $f$ until value doesn't change:
  $x \rightarrow f(x) \rightarrow f(f(x)) \rightarrow f(f(f(x))) \rightarrow \ldots$



## Fixed Points

```
fixedPoint :: (Float -> Float) -> Float -> Float
fixedPoint f x0 = fpIter x0
  where
    fpIter :: Float -> Float
    fpIter x
      | isCloseEnough x x' = x'
      | otherwise          = fpIter x'
      where
        x' = f x

isCloseEnough :: Float -> Float -> Bool
isCloseEnough x x' = (abs (x' - x) / x) < 0.001
```



## Square Roots

use fixed points to compute square roots

- $y = \sqrt{x} \Rightarrow y * y = x \Rightarrow y = x/y$
- fixed point of the function $f(y) = x/y$

```
sqrt :: Float -> Float
sqrt x = fixedPoint (\y -> x / y) 1.0
```

- doesn't converge (try with $x = 2.0$)



## Square Roots

- average successive values (average damping)

```
sqrt x = fixedPoint (\y -> (y + x/y) / 2.0) 1.0
```

- exercise: implement average damping as a higher order function and use it in sqrt implementation

## Filter

- select all elements with a given property

all odd elements of a list

```
-- allOdds [4, 1, 3, 2] ~> [1, 3]

allOdds :: [Integer] -> [Integer]
allOdds []     = []
allOdds (x:xs)
  | odd x      = x : allOdds xs
  | otherwise  = allOdds xs
```



## Filter

- `filter`: select elements that satisfy a predicate

```
filter f []     = []
filter f (x:xs)
  | f x         = x : filter f xs
  | otherwise   = filter f xs
```

- what is the type of `filter`?
  `filter :: (a -> Bool) -> [a] -> [a]`



## Filter Example

all odd elements of a list

```
allOdds :: [Integer] -> [Integer]
allOdds xs = filter odd xs
```

Python

```
filter(lambda x: x % 2 == 1, [4, 1, 3, 2])
```



## Filter Example

how many elements in a list are above a threshold?

```
howManyAbove :: Float -> [Float] -> Int
howManyAbove t xs = length (filter (\x -> x >= t) xs)
```

## Splitting Lists

- take elements from the front of a list while a predicate is true
  `takeWhile even [8, 2, 4, 5, 6] ~> [8, 2, 4]`

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile f []     = []
takeWhile f (x:xs)
  | f x       = x : takeWhile f xs
  | otherwise = []
```

- exercise: drop elements from the front of a list while a predicate is true
  `dropWhile even [8, 2, 4, 5, 6] ~> [5, 6]`



## Map

- transform all elements of a list

example: floors of all elements of a list

```
-- floorAll [5.7, 9.0, 2.3] ~> [5, 9, 2]

floorAll :: [Float] -> [Integer]
floorAll []     = []
floorAll (x:xs) = floor x : floorAll xs
```



## Map

- `map`: apply a function to all elements of a list

```
map f []     = []
map f (x:xs) = f x : map f xs
```

- what is the type of `map`?
  `map :: (a -> b) -> [a] -> [b]`



## Map Example

floors of all elements of a list

```
floorAll :: [Float] -> [Integer]
floorAll xs = map floor xs
```

Python

```
from math import floor

map(floor, [5.7, 9.0, 2.3])
```

## Map Examples

make a list of n copies of an item

```
replicate :: Int -> a -> [a]
replicate n i = map (\_ -> i) [1 .. n]
```

zip two lists over a function

```
zipWith (+)       [1, 2] [10, 12]   ~> [11, 14]
zipWith replicate [3, 2] ['a', 'b'] ~> ["aaa", "bb"]

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f xs ys = map (\(x, y) -> f x y) (zip xs ys)
```



## Fold

- reduce the elements of a list to a single value

sum all elements of a non-empty list

```
-- sum [2, 8, 5] ~> 15

sum :: [Integer] -> Integer
sum [x]    = x
sum (x:xs) = x + sum xs
```



## Fold

- `foldr1`: reduce a non-empty list to a value over a function

```
foldr1 f [x]    = x
foldr1 f (x:xs) = x `f` (foldr1 f xs)

-- OR:
foldr1 f [x]    = x
foldr1 f (x:xs) = f x (foldr1 f xs)
```

- what is the type of `foldr1`?
  `foldr1 :: (a -> a -> a) -> [a] -> a`



## Fold Expansion

```
foldr1 f [e1, e2, ..., ej, ek]
    = f e1 (foldr1 f [e2, ..., ej, ek])
    = e1 `f` (foldr1 f [e2, ..., ej, ek])
    = e1 `f` (e2 `f` (... (ej `f` ek)...)
```

# Fold

sum all elements of a list

```
sum :: [Integer] -> Integer
sum xs = foldr1 (+) xs
```

Python

```
from functools import reduce
from operator import add

def sum(xs):
    return reduce(add, xs)
```



# Fold with Initial Value

- `foldr1` doesn't work on empty lists
- add a parameter as initial value for empty list: `foldr`

```
foldr f s []     = s
foldr f s (x:xs) = f x (foldr f s xs)
```

- what is the type of `foldr`?
  `foldr :: (a -> b -> b) -> b -> [a] -> b`



# Fold with Initial Value Expansion

```
foldr f s [e1, e2, ..., ej, ek]
    = f e1 (foldr f s [e2, ..., ej, ek])
    = e1 ‘f‘ (foldr f s [e2, ..., ej, ek])
    = e1 ‘f‘ (e2 ‘f‘ (... (ej ‘f‘ (ek ‘f‘ s))...)
```



# Fold with Initial Value

sum all elements of a list

```
sum :: [Integer] -> Integer
sum xs = foldr (+) 0 xs
```

Python

```
from functools import reduce
from operator import add

def sum(xs):
  return reduce(add, xs, 0)
```

## Fold Examples

```
product :: [Integer] -> Integer
product xs = foldr (*) 1 xs

fac :: [Integer] -> Integer
fac n = foldr (*) 1 [1 .. n]

and :: [Bool] -> Bool
and xs = foldr (&&) True xs

concat :: [[a]] -> [a]
concat xs = foldr (++) [] xs

maxList :: [Integer] -> Integer
maxList xs = foldr1 max xs
```



## Fold Example

how many elements in a list are above a threshold?

```
howManyAbove :: Float -> [Float] -> Integer
howManyAbove t xs =
    foldr (\x n -> if x >= t then n + 1 else n) 0 xs
```



## Fold Example

insertion sort

```
ins :: Integer -> [Integer] -> [Integer]
ins n []          = [n]
ins n xs@(x':xs')
  | n <= x'       = n : xs
  | otherwise     = x' : ins n xs'

iSort :: [Integer] -> [Integer]
iSort []     = []
iSort (x:xs) = ins x (iSort xs)

-- equivalent to:
iSort :: [Integer] -> [Integer]
iSort xs = foldr ins [] xs
```



## Fold Left

```
foldl f s [e1, e2, ..., ej, ek]
    = (...((s `f` e1) `f` e2) `f` ... ej) `f` ek
    = foldl f (s `f` e1) [e2, ..., ej, ek]
    = foldl f (f s e1) [e2, ..., ej, ek]
```

```
foldl f s []     = s
foldl f s (x:xs) = foldl f (f s x) xs
```

- what is the type of `foldl`?
  `foldl :: (b -> a -> b) -> b -> [a] -> b`

## Fold Right - Fold Left

- results not the same if function is not commutative

example

```
foldr (*) 1   [3 .. 6]        ~> 360
foldl (*) 1   [3 .. 6]        ~> 360

foldr (/) 6.0 [3.0, 2.0, 4.0] ~> 1.0
foldl (/) 6.0 [3.0, 2.0, 4.0] ~> 0.25
```



## Edit Distance

transform a source string into a destination string

- operations: copy, insert, delete, change
- costs: copy 0, all others 1
- find path with minimum cost

```
data Edit = Copy | Insert Char | Delete | Change Char
            deriving (Eq, Show)
```



## Edit Distance

```
transform :: String -> String -> [Edit]
transform [] [] = []
transform xs [] = map (\_ -> Delete) xs
transform [] ys = map Insert ys
transform xs@(x':xs') ys@(y':ys')
  | x' == y'  = Copy : transform xs' ys'
  | otherwise = best [Insert y' : transform xs  ys',
                      Delete    : transform xs' ys,
                      Change y' : transform xs' ys']
```



## Edit Distance

find best path

```
best :: [[Edit]] -> [Edit]
best [x]    = x
best (x:xs)
  | cost x <= cost b = x
  | otherwise        = b
  where
    b = best xs

cost :: [Edit] -> Int
cost xs = length (filter (\x -> x /= Copy) xs)
```

- exercise: implement `best` using `fold`

## List Comprehension

- describe a list in terms of the elements of another list
- generate, test, transform

```
[e | v1 <- l1, v2 <- l2, ..., p1, p2, ...]
```



## List Comprehension Examples

```
[2 * n | n <- [2, 4, 7]]  ~> [4, 8, 14]

[even n | n <- [2, 4, 7]] ~> [True, True, False]

[2 * n | n <- [2, 4, 7], even n, n > 3] ~> [8]

[m + n | (m, n) <- [(2, 3), (2, 1), (7, 8)]]
~> [5, 3, 15]

[(x, y, z) | x <- [1 .. 5], y <- [1 .. 5],
             z <- [1 .. 5],
             x*x + y*y == z*z]
```



## List Comprehension Examples

Python

```
[2 * n for n in [2, 4, 7]]

[n % 2 == 0 for n in [2, 4, 7]]

[2 * n for n in [2, 4, 7] if (n % 2 == 0) and (n > 3)]

[m + n for (m, n) in [(2, 3), (2, 1), (7, 8)]]

[(x, y, z) for x in range(1, 6)
           for y in range(1, 6)
           for z in range(1, 6)
           if x * x + y * y == z * z]
```



## List Comprehension Example

quick sort

```
qSort :: [Integer] -> [Integer]
qSort []     = []
qSort (x:xs) =
    qSort smaller ++ [x] ++ qSort larger
      where
        smaller = [a | a <- xs, a <= x]
        larger  = [b | b <- xs, b >  x]
```

## Higher Order List Functions

```
filter f xs = [x | x <- xs, f x]

map f xs = [f x | x <- xs]
```



## Python Comprehensions

- list comprehension: `[x for ...]`

```
[x * x for x in [2, 4, 7, -2]]
~> [4, 16, 49, 4]
```

- set comprehension: `{x for ...}`

```
{x * x for x in [2, 4, 7, -2]}
~> {4, 16, 49}
```

- dictionary comprehension: `{k: v for ...}`

```
{s: len(s) for s in ['haskell', 'python', 'foo']}
~> {'haskell': 7, 'python': 6, 'foo': 3}
```



## References

Required Reading: Thompson

- Chapter 10: Generalization: patterns of computation