Functional Programming

Match Case

Kasun de Zoysa
kasun@ucsc.cmb.ac.lk

UNIVERSITY OF COLOMBO SCHOOL OF COMPUTING

Problems

1. Define the Scala function **maximum**, which take two integers and determine the maximum number.

2. Write a Scala function **passfail** which print a student grade as follows:
Marks >= 50 "Pass"
Else "Fail"

3. Define the Scala function **iseven.** It checks if a numeric value is an even number or not**.**

maximum(a,b)

```
scala> def maximum(a:Int,b:Int):Int=
if (a>=b) a else b
maximum: (a: Int, b: Int)Int


scala> maximum(2,5)
res1: Int = 5


scala> maximum(12,8)
res7: Int = 12
```

match

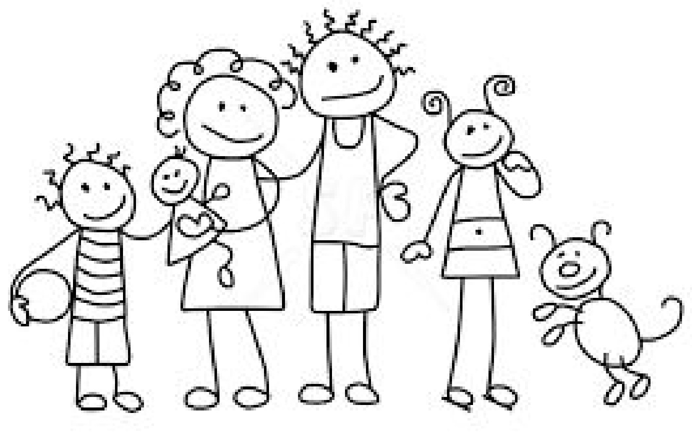
Scala’s match expressions are an amazingly flexible device that also enables matching diverse items as types, regular expressions, numeric ranges, and data structure contents.

'If he could trace the matching sock I've another 25 or 30 to account for.'

match

Scala developers prefer match expressions over “if .. else” blocks because of their expressiveness and concise syntax.

```
<expression> match {
case <pattern match> => <expressic
[case...]
}
```



maximum(a,b)

```
scala> def maximum(a:Int,b:Int):Int= a>=b match{
     | case true => a
     | case false => b
     | }
maximum: (a: Int, b: Int)Int


scala> maximum(10,8)
res8: Int = 10
scala> maximum(1,8)
res9: Int = 8
```

passfail(marks)

The Scala syntax looks like this:

```
scala> def
passfail(marks:Int):String=
     | marks>=50 match{
     | case true => "PASS"
     | case false=> "FAIL"
     | }
```

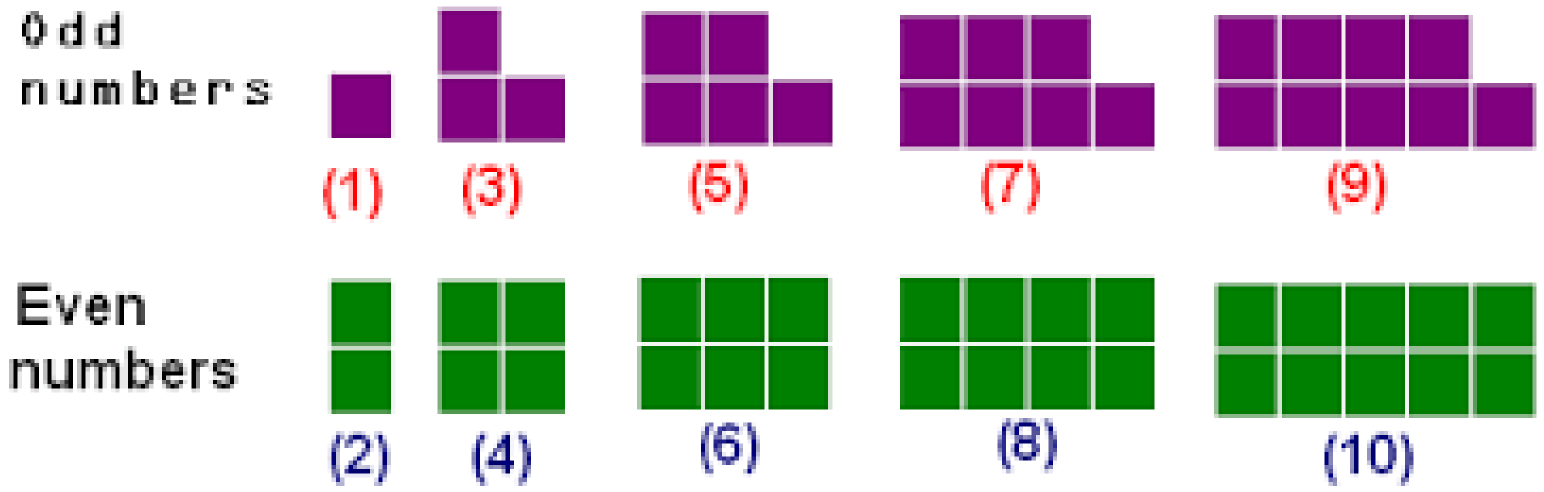
Matching with Wildcard Patterns

- There are two kinds of wildcard patterns:
  - Value binding
  - wildcard (aka “underscore”) operators

- The value **other** is defined for the duration of the case block and is assigned the value of the input to the match expression.

- Wildcard pattern is an underscore (_) character that acts as an unnamed placeholder for the eventual value of an expression at runtime.

iseven(number)



iseven(number)

The Scala syntax looks like this:

```
scala> def iseven(x:Int):Boolean=
     | x%2 match{
     | case 0 => true
     | case _ => false
     | }
```

Functions that Test Several Conditions

1. Write a program to check leap year.

Leap year:

- If the year is exactly divisible by 4 and not divisible by 100 then its Leap Year
- else if the year is exactly divisible by 400 then its Leap Year
- else its a common year.

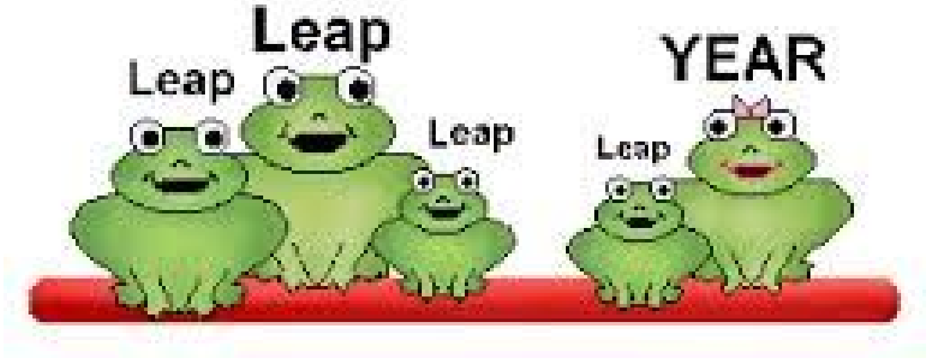
2.Write a function `sign` that return correct value

$$sign(x) = \begin{cases} +1 \text{ if } x > 0 \\ 0 \text{ if } x = 0 \\ -1 \text{ if } x < 0 \end{cases}$$

leapYear(year)

The Scala syntax looks like this:

```
scala> def leapYear(year:Int):Boolean=
     | (year%4==0 && year%100!=0) || year%400==0
match{
     | case true => true
     | case false => false
     | }
```



Chained Conditions - Scala

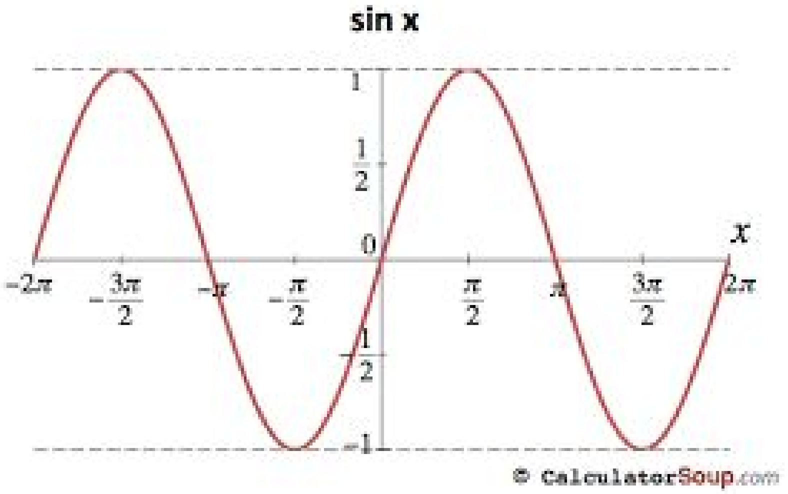
Multiple if...else statements can be nested to create an else if clause.

```
scala> def sign(x:Int):Int=

|if (x>0) 1 else if (x==0) 0 else -1

sign: (x: Int)Int
```

```
scala> sign(-3)
res0: Int = -1
scala> sign(0)
res1: Int = 0
scala> sign(1)
res2: Int = 1
```



Matching with Pattern Guards

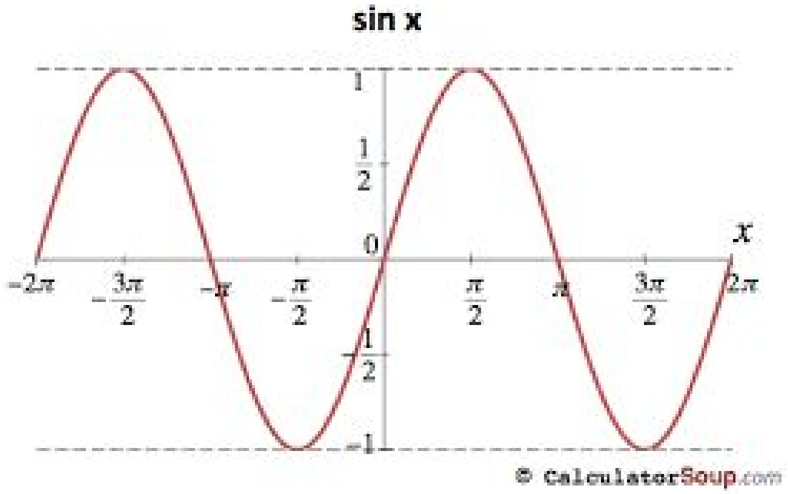
A pattern guard adds an if expression to a value-binding pattern, making it possible to mix conditional logic into match expressions. When a pattern guard is used the pattern will only be matched when the if expression returns true.

```
case <pattern> if <Boolean expression> =>

<one or more expressions>
```

You can define this **sign** function in Scala using a **match** expression:

```
scala> def sign(x:Int):Int= x match{
     | case x if x>0 => 1
     | case x if x==0 => 0
     | case x if x<0 => -1
     | }
```



Student Grade

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a chained conditional .

Write a function **grade** which print a student grade as follows:

```
Marks >= 75 => "A"
Marks >= 65 => "B"
Marks >= 50 => "C"
Else "F"
```

grade(mark)

```
scala> def grade(mark:Int)= mark match{
     | case x if x>=75 => "A"
     | case x if x>=65 => "B"
     | case x if x>=50 => "C"
     | case _ => "F"
     | }
grade: (mark: Int)String


scala> grade(5)
res15: String = F
scala> grade(50)
res16: String = C
```

Problems

1. Develop the function **interest**. It consumes a deposit amount and produces the actual amount of interest that the money earns in a year.

The bank pays a flat 5% for deposits of up to Rs. 1000, a flat 6% per year for deposits of up to Rs. 10000, a flat 7% per year for deposits of up to Rs. 100000, and a flat 8% for deposits of more than Rs. 100000.

2. Develop the function **balance**. It consumes a deposit amount and produces the final account balance

interest(amount)

```
scala> def interest(amount:Double):Double=

amount match {

     | case x if x<0 => 0

     | case x if x<1000 => x*.05

     | case x if x<10000 => x*.06

     | case x if x<100000 => x*.07

     | case x if x>=100000 => x*.08

     | }
```

```
scala> interest(100)
res1: Double = 5.0
scala> interest(98790)
res2: Double = 6915.300000000001
```

•

Take Home Salary

Develop the function **tax**, which consumes the gross pay and produces the amount of tax owed.

For a gross pay of Rs.50000 or less, the tax is 0%;
for gross pay over Rs. 50000 and Rs.100000 or less, the tax rate is 10%;
and for any pay over Rs. 100000, the tax rate is 20%.

Also develop a function **netpay**. The function determines the net pay of an employee from the number of hours worked. The net pay is the gross pay minus the tax. Assume the hourly pay rate is Rs. 500.

Take Home Salary

```
val rate:Double=500.00

def income(h:Int):Double=h*rate

def tax(income:Double):Double=income match{
case x if x<= 50000 => 0
case x if x<=100000 => x*.1
case x if x>100000  => x*.2
}

def netpay(hours:Int):Double = income(hours) -
tax(income(hours))
```

Discussion