Ficha 5

Exercício 5.1

a)

```
def maximum(tree: MyTree[Int]): Option[Int] = tree match {
 case Empty => None
 case Node(value, left, right) => {
  val leftMax = maximum(left)
  val rightMax = maximum(right)
  (leftMax, rightMax) match {
   case (Some(l), Some(r)) => Some(value.max(l).max(r)) //Some(value max l max r)
   case (Some(l), None) => Some(value.max(l))
   case (None, Some(r)) => Some(value.max(r))
   case (None, None) => Some(value)
  }
 }
}
```

b)

```
def depth[A](t: MyTree[A]): Int = t match {
   case Empty => 0
   case Node(_,l,r) => 1 + (depth(l) max depth(r))
}
```

c)

```
def map[A,B](t: MyTree[A])(f: A => B): MyTree[B] = t match {
   case Empty => Empty
   case Node(v,l,r) => Node(f(v),map(l)(f), map(r)(f))
}
```

Exercício 5.2

a)

```
def trabs(t:Turma): Turma = {
   t.alunos filter(x=>x._3 == RegimeOPT.TrabEstud)
}
```

OU

```
def trabs1(t: Turma): Alunos = {
   def aux(l: Alunos): Alunos = l match {
      case Nil => Nil
   case x::xs => if(x._3 == RegimeOPT.TrabEstud) x::aux(xs) else aux(xs)
   }
   aux(t.alunos)
}
```

b)

```
def searchStudent(t: List[Aluno], n: Int) : Option[Aluno] = {
   (t foldRight [Option[Aluno]](None))((x:Aluno, tail) =>
    if(x._1 == n)
      Some(x)
    else
   tail)
}
```

OU

```
def searchStudent(t: List[Aluno], n: Int) : Option[Aluno] = {
  t.foldRight(None:Option[Aluno])((x:Aluno, tail) =>
    if(x._1 == n)
      Some(x)
    else
   tail)
}
```

OU

```
def searchStudent1(t: Alunos, n: Numero) : Option[Aluno] = {
  t match {
    case Nil => None
    case x::xs => if(x._1 == n) Some(x) else searchStudent1(xs, n)
  }
}
```

c)

```
//utilizando alinea anterior
def finalGrade1(n: Numero, t: Turma): Option[Float] = {

  val al = searchStudent1(t.alunos, n)

  if (al != None) {
      if (al.get._4 != None && al.get._5 != None && al.get._4.get >= 9.5
&& al.get._5.get >= 9.5)
      Some(al.get._4.get * 0.6.toFloat + al.get._5.get * 0.4.toFloat)
    else None
  }
  else
  None
}
```

OU

```
def finalGrade(n: Numero, t: Turma): Option[Float] = {

  val index = t.alunos.indexWhere(x => { x._1 == n })

  if (index != -1) {
    val al = t.alunos.apply(index)
    if (al._4 != None && al._5 != None && al._4.get >= 9.5 && al._5.get
>= 9.5)
      Some(al._4.get * 0.6.toFloat + al._5.get * 0.4.toFloat)
    else None
  }
  else
   None
}
```

d)

```
def approved(t: Turma): List[(Nome, Float)] = {
  val t1 = t
  (t.alunos foldRight List[(Nome,Float)]()) ((x,t) => {
    val r = finalGrade(x._1,t1)
    if(r.nonEmpty && r.get >= 10) (x._2,r.get)::t
    else t
  })
}
```

e)

```
def changeNP(n: Numero, np: NP, t: Turma): Turma = {
  val index = t.alunos.indexWhere(x => { x._1 == n } )
  if(index != -1)
  {
    val al = t.alunos.apply(index)
    val al1 = (al._1, al._2, al._3, al._4, np)
    val t1 = new Turma(t.id, t.alunos.updated(index, al1))
    t1
  }
  else t
}

def changeNT(n: Numero, nt: NT, t: Turma): Turma = {
  val index = t.alunos.indexWhere(x => { x._1 == n } )
  if(index != -1)
  {
    var al = t.alunos.apply(index)
    al = (al._1, al._2, al._3, nt, al._5)
    val t1 = new Turma(t.id, t.alunos.updated(index, al))
    t1
  }
  else t
}
```

f)

```
def insertOrd(a:Aluno, t1:Turma) : Turma = {
  val t = t1.alunos
  new Turma(t1.id, (t foldRight List[Aluno](a)) ((x,t) => if( a._1 < x._1)
a::x::t else x::t))
}
```

OU

```
def insertOrd1(a:Aluno, t:Turma) : Turma = {
  t match{
    case Nil => List(a)
    case x::xs => if( a._1 < x._1) a::x::xs else x::insertOrd1(a, xs)
  }
}
```

g)

```
def searchStudentOrd(n: Numero, t1: Turma):Option[Aluno] = {
  val t = t1.alunos
  (t foldRight None) ((x,xs) => if(x._1 == n) Some(x) else if(x._1 < n) xs
else None)
}
```

OU

```
def searchStudentOrd(t1: Turma, n: Numero) : Option[Aluno] = {
  def aux(t :List[Aluno], n:Numero):Option[Aluno] = {
    t match {
      case Nil => None
      case x::xs => if(x._1 == n) Some(x) else if(x._1 < n) aux(xs, n) else
None
    }
  }
  aux(t1.alunos,n)
}
```

Exercício 5.3

a)

```
def trabsTree(t: MyTree[Aluno]):List[Aluno]={
  t match {
    case Empty => Nil
  case Node(a, left, right) =>
      if (a._3 == RegimeOPT.TrabEstud)
        a :: trabsTree(left) ++ trabsTree(right)
      else
    trabsTree(left) ++ trabsTree(right)
  }
}
```

b)

```
def searchStudentTree(n:Numero, t:MyTree[Aluno]):Option[Aluno]={
  t match {
    case Empty => None
    case Node(a, left, right) =>
      if (n < a._1)
```

```
        searchStudentTree(n,right)
      else if (n > a._1)
        searchStudentTree(n, left)
      else
    Some(a)
  }
}
```

c)

```
def finalGradeTree(n: Numero, t:MyTree[Aluno]):Option[Float]={
  t match {
    case Empty => None
    case Node(a, left, right) =>
      if (n<a._1)
        finalGradeTree(n,right)
      else if (n>a._1)
        finalGradeTree(n,left)
      else
    if (a._4 != None && a._5 != None && a._4.get >= 9.5 &&
a._5.get >= 9.5)
          Some(a._4.get*0.6f+a._5.get*0.4f)
        else
     None
  }
}
```

d)

```
def approvedTree(t: MyTree[Aluno]):List[(Nome,Float)] ={
  t match {
    case Empty => Nil
  case Node(a, left, right) =>{
      val r= finalGradeTree(a._1,t)
      if(r.nonEmpty && r.get>=10)
        (a._2, r.get):: approvedTree(left)++ approvedTree(right)
      else
    approvedTree(left) ++ approvedTree(right)
    }
  }
}
```

e)

```
def changeNPTree(n: Numero, np:
NP,t:MyTree[Aluno]):MyTree[Aluno]={
t match {
  case Empty => Empty
  case Node(a, left, right) =>
      if (n< a._1)
```

```
          Node(a,left,changeNPTree(n,np,right))
        else if (n > a._1)
          Node(a,changeNPTree(n,np,left),right)
        else
      Node((a._1,a._2,a._3,a._4,np),left,right)
    }
}

def changeNTTree(n: Numero, nt: NT,
t:MyTree[Aluno]):MyTree[Aluno]={
  t match {
    case Empty => Empty
    case Node(a, left, right) =>
      if (n< a._1)
        Node(a,left,changeNTTree(n,nt,right))
      else if (n > a._1)
        Node(a,changeNTTree(n,nt,left),right)
      else
    Node((a._1,a._2,a._3,nt, a._5),left,right)
  }
}
```

f)

```
def insertTree(a: Aluno, t:MyTree[Aluno]):MyTree[Aluno]={
  t match {
    case Empty => Node(a,Empty, Empty)
    case Node(v, left, right) =>
      if(a._1 < v._1)
        Node(v, left, insertTree(a, right))
      else if (a._1 > v._1)
        Node(v, insertTree(a,left),right)
      else
    t
  }
}
```