1. Data je hijerarhija klasa za realizaciju ulančane liste sa parametrizovanim tipom elementa (liste.scala).
   a) Napisati funkciju koja dohvata n-ti element liste.
      `def ntiElement[T](n: Int, lista: List[T]): T`
   b) Napisati funkciju koja odbacuje prvih n elemenata liste.
      `def izbaciN[T](n: Int, lista: List[T]): List[T]`
   c) Napisati funkciju koja odbacuje elemente sa početka liste koji ispunjavanju zadati uslov.
      `def izbaciDok[T](lista: List[T], p: T => Boolean): List[T]`

2. Dopuniti sledećim operatorima datu hijerarhiju klasa za realizaciju logičkog tipa (`Boolean`) bez upotrebe kontrolne strukture `if` (boolean.scala):
   a) `==`
   b) `&&`
   c) `||`
   d) `<`

3. Data je hijerarhija klasa za rad sa skupovima celih brojeva (intset.scala). Proširiti klase iz `IntSet` hijerarhije metodama za određivanje unije i preseka skupova.

```scala
//liste.scala
object list extends App {

  abstract class List[T] {
    def prazna: Boolean
    def glava: T
    def rep: List[T]
  }

  class Nil[T] extends List[T] {
    def prazna = true
    def glava = throw new NoSuchElementException("Nil.glava")
    def rep = throw new NoSuchElementException("Nil.rep")
  }

  class Elem[T](val glava : T, val rep : List[T]) extends List[T] {
    def prazna = false
  }

  val testList = new Elem(1, new Elem(2, new Elem(3, new Nil)))


  //print(ntiElement(1, testList))
  //print(izbaciN(1, testList).glava)
  //print(izbaciDok(testList, (x: Int) => x<=2).glava)

}
```

```scala
//boolean.scala
object boolean extends App {

  abstract class Boolean {
    def ifThenElse(t : => Boolean, f : => Boolean) : Boolean
    def unary_! : Boolean = ifThenElse(False, True)
    def != (x : => Boolean) : Boolean = ifThenElse(!x, x)
  }

  object True extends Boolean {
    def ifThenElse(t : => Boolean, f : => Boolean) : Boolean = t
  }

  object False extends Boolean {
    def ifThenElse(t : => Boolean, f : => Boolean) : Boolean = f
  }

  val a = True
  val b = False

  //print(a == b)
  //print(a || b)
  //print(a && b)
  //print(a < b)
}
```

```scala
//intset.scala
object intset extends App {

  abstract class IntSet {
    def incl(x: Int): IntSet
    def contains(x: Int): Boolean
  }

  class Empty extends IntSet {
    def contains(x : Int) = false
    def incl(x : Int) =
      new NonEmpty(x, new Empty, new Empty)
  }

  class NonEmpty(elem: Int, left: IntSet, right: IntSet) extends IntSet {
    def this(elem: Int) = this(elem, new Empty, new Empty)
    def contains(x : Int) = {
      if (x < elem) left contains x
      else if (x > elem) right contains x
      else true
    }
    def incl(x : Int) = {
      if (x<elem) new NonEmpty(elem, left incl x, right)
      else if (x>elem) new NonEmpty(elem, left, right incl x)
      else this
    }
  }

  val x = new NonEmpty(10, new Empty, new Empty)

}
```