Profs. Viktor Kunčak, Martin Odersky, and Clément Pit-Claudel
CS-214 Software Construction make-up midterm
01.11.2023 from 16:15 to 17:45
Duration: 90 minutes

1

SCIPER : **1000001** ROOM : **SG 1**

Annie Easley

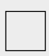
Wait for the start of the exam before turning to the next page. This document is printed double sided, 16 pages. Do not unstaple.

Material	This is a closed book exam. Paper documents and electronic devices are not allowed. Place on your desk your student ID and writing utensils. Place all other personal items at the front of the room. If you need additional draft paper, raise your hand and we will provide some.
Time	All points are not equal: we do not think that all exercises have the same difficulty, even if they have the same number of points. Manage your time accordingly. You may want to look at the whole exam before starting on a particular exercise.
Appendix	The last page of this exam contains an appendix which is useful for formulating your solutions. Do not detach this sheet.
Use a pen	For technical reasons, **only use black or blue pens for the MCQ part, no pencils!** Use white corrector if necessary.
Grading Scheme	The exam contains a total of 100 points. For multiple choice questions, a good answer is worth 4 points and a bad answer 0 points. Note that there is always exactly one good answer to each question. For true-false questions, a good answer is worth 2 points and a bad answer 0 points. For open questions, the number of points is variable and indicated at the top of each question.
Stay Functional	Do not use `var`s, `while` loops, `for...do` loops, etc. This will result in 0 points for that question.

Respectez les consignes suivantes | Read these guidelines | Beachten Sie bitte die unten stehenden Richtlinien

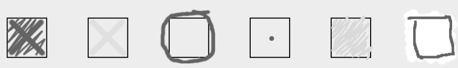
choisir une réponse | select an answer
Antwort auswählen

ne PAS choisir une réponse | NOT select an answer
NICHT Antwort auswählen



Corriger une réponse | Correct an answer
Antwort korrigieren

ce qu'il ne faut **PAS** faire | what should **NOT** be done | was man **NICHT** tun sollte



Run-length encoding (11 pts)

Question 1 This question is worth 11 points.

☐ 0 ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ 8 ☐ 9 ☐ 10 ■ 11 *Do not write here.*

Run-length encoding is a simple compression technique that replaces contiguous sequences of equal elements in a list by a pair containing the element and a number indicating how many times it was repeated.

Write a function `runLengthEncode[T](xs: List[T]): List[(T, Int)]` that takes a list of elements and returns a run-length-encoded version of the input.

Here are example tests that your implementation must pass:

```
test("runLengthEncode: empty list"):
  assertEquals(runLengthEncode(Nil), Nil)

test("runLengthEncode: list without repeated elements"):
  assertEquals(runLengthEncode(List("a", "b")), List(("a", 1), ("b", 1)))

test("runLengthEncode: list with repeated elements"):
  assertEquals(
    runLengthEncode(List("x", "a", "a", "x")),
    List(("x", 1), ("a", 2), ("x", 1))
  )
```

The runtime complexity of your implementation should not be more than linear ($\mathcal{O}(n)$).

```
def runLengthEncode[T](xs: List[T]): List[(T, Int)] =
  xs match
    case Nil => Nil
    case h :: t =>
      runLengthEncode(t) match
        case (`h`, n) :: rlt => (h, n + 1) :: rlt
        case rlt             => (h, 1) :: rlt
```

Mystery function (10 pts)

Question 2 This question is worth 10 points.

☐ 0 ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ 8 ☐ 9 ■ 10 *Do not write here.*

In this exercise, your task is to use the substitution method to write the step-by-step evaluation of an expression, under the call-by-value evaluation strategy.

You must apply the definition of a single function call at a time and write the result of each step. You can directly reduce if-then-else expressions to their branches.

As an example, consider the function `factorial`:

```
def factorial(n: Int): Int =
  if n == 0 then 1
  else n * factorial(n - 1)
```

The expression `factorial(2)` evaluates step-by-step as follows:

```
    factorial(2)
=== 2 * factorial(1)
=== 2 * (1 * factorial(0))
=== 2 * (1 * 1)
=== 2 * 1
=== 2
```

Now, consider the function `f`:

```
def f(x: Int, y: Int, z: Int = 0): Int =
  if x < y && z == 0 then 0
  else if z == 0 then 1 + f(x - y, y, y) - y
  else 1 + f(x, y, z - 1)
```

Write the step-by-step evaluation of the expression `f(3, 3)`:

```
     f(3, 3, 0)
=== (1 +  f(0, 3, 3)                - 3)
=== (1 + (1 +  f(0, 3, 2)           ) - 3)
=== (1 + (1 + (1 +  f(0, 3, 1)      )) - 3)
=== (1 + (1 + (1 + (1 + f(0, 3, 0)))) - 3)
=== (1 + (1 + (1 + (1 + (0)       ))) - 3)
=== 1
```

What does `f(x, y)` compute when `x` and `y` are positive, in a few words?

```
The integer division x / y
```

Permutations (14 pts)

A sequence `xs: Seq[Int]` defines a function $f : i \mapsto$ `xs(i)`. If this function is a bijection from $[0,$ `xs.length`$)$ into $[0,$ `xs.length`$)$, we call the sequence a *permutation*.

As a reminder, a function $f : A \to B$ is a bijection if each element of A and B is paired with exactly one element of the other set.

For example, `Seq(0, 3, 1, 2)` is a permutation, and so is `Seq(0, 1, 2, 3)`.

Given below are 7 different implementations of the `isPermutation` function. A correct implementation must return **`true`** if the given sequence is a permutation, or **`false`** otherwise. For each implementation, tick "Yes" if it is correct (for all possible inputs), or "No" if it is incorrect.

```scala
def isPermutation1(xs: Vector[Int]): Boolean =
  (0 until xs.length).forall(xs.contains)
```

Question 3 Is `isPermutation1` correct?

■ Yes ☐ No

```scala
def isPermutation2(xs: Vector[Int]): Boolean =
  def loop(xs: Vector[Int], ys: Set[Int]): Boolean =
    if xs.isEmpty then true
    else
      ys.contains(xs.head) &&
      loop(xs.tail, ys - xs.head)
  loop(xs, xs.toSet)
```

Question 4 Is `isPermutation2` correct?

☐ Yes ■ No

```scala
def isPermutation3(xs: Vector[Int]): Boolean =
  xs.toSet.size == xs.size
```

Question 5 Is `isPermutation3` correct?

☐ Yes ■ No

```scala
def isPermutation4(xs: Vector[Int]): Boolean =
  xs.forall(x => xs.count(_ == x) == 1)
```

Question 6 Is `isPermutation4` correct?

☐ Yes ■ No

```
def isPermutation5(xs: Vector[Int]): Boolean =
  def loop(ys: Vector[Int]): Boolean =
    if ys.isEmpty then true
    else
      0 ≤ ys.head &&
      ys.head < xs.length &&
      xs.count(_ == ys.head) == 1
  loop(xs)
```

Question 7 Is `isPermutation5` correct?

☐ Yes ■ No

```
def isPermutation6(xs: Vector[Int]): Boolean =
  def loop(xs: Vector[Int], ys: Set[Int]): Set[Int] =
    if xs.isEmpty then ys
    else loop(xs.tail, ys + xs.head)
  loop(xs, Set()) == (0 until xs.length).toSet
```

Question 8 Is `isPermutation6` correct?

■ Yes ☐ No

```
def isPermutation7(xs: Vector[Int]): Boolean =
  xs.reverse == xs
```

Question 9 Is `isPermutation7` correct?

☐ Yes ■ No

Proof of ContainsConcat (12 pts)

Question 10 This question, consisting of both cases of the proof, is worth 12 points.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6 ☐7 ☐8 ☐9 ☐10 ☐11 ■12 *Do not write here.*

All lemmas on this page hold for all types `T` and all `x: T`, `y: T`, `b1: Boolean`, `b2: Boolean`, `b3: Boolean`, `xs: List[T]`, `ys: List[T]`, `l: List[T]`, `m: List[T]`.

Given the following lemmas:

(CONCATNILL) `Nil ++ xs === xs`

(CONCATNILR) `xs ++ Nil === xs`

(CONCATCONS) `(x::xs) ++ ys === x::(xs ++ ys)`

(CONTAINSNIL) `Nil.contains(x) === false`

(CONTAINSCONS) `(x :: xs).contains(y) === x == y | xs.contains(y)`

(ORASSOC) `b1 | (b2 | b3) === (b1 | b2) | b3`

(ORCOMM) `b1 | b2 === b2 | b1`

(ORFALSEL) `b === false | b`

(ORFALSER) `b === b | false`

You need to prove:

(CONTAINSCONCAT) `(l ++ m).contains(y) === l.contains(y) | m.contains(y)`

Complete the proof below. For each step, you must write the name of the lemma you are using. You may only use the lemmas above.

The proof is done by induction on `l`.

Base case: `l` is `Nil`. Therefore, you need to prove:

`(Nil ++ m).contains(y) === Nil.contains(y) | m.contains(y)`

```
    (Nil ++ m).contains(y)
=== m.contains(y)                   // by ConcatNilL
=== False | m.contains(y)           // by OrFalseL
=== Nil.contains(y) | m.contains(y) // by ContainsNil
```

Induction step: `l` is `x :: xs`. Therefore, you need to prove:

```
((x::xs) ++ m).contains(y) === (x::xs).contains(y) | m.contains(y)
```

given that the induction hypothesis, named IH, holds:

(IH) `(xs ++ m).contains(y) === xs.contains(y) | m.contains(y)`

```
    ((x :: xs) ++ m).contains(y)
=== (x :: xs ++ m).contains(y)                   // by ConcatCons
=== (y == x) | (xs ++ m).contains(y)             // by ContainsCons
=== (y == x) | (xs.contains(y) | m.contains(y)) // by IH
=== ((y == x) | xs.contains(y)) | m.contains(y) // by OrAssoc
=== (x :: xs).contains(y) | m.contains(y)       // by ContainsCons
```

`for` Comprehension (8 pts)

Question 11 This question is worth 8 points.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6 ☐7 ■8 *Do not write here.*

The *abundancy* of a number is the ratio of the sum of its divisors to itself. For example, the abundancy of `30` is $\mathrm{a}(30) = \frac{1+2+3+5+6+10+15+30}{30} = \frac{72}{30} = \frac{12}{5}$

A *friendly pair* consists of two positive integers (a, b) with the same abundancy. For example, $(30, 140)$ is a friendly pair because $\mathrm{a}(30) = \mathrm{a}(140)$.

Implement a function `friendly(n: Int)` takes an integer $n < 10^4$ as a parameter and produces a list of all friendly pairs (a, b) such that $0 < a < b \leq n$, in at most $\mathcal{O}(n^3)$ time.

The list should have no duplicates.

You must use a `for` comprehension in order to get any points for this question.

```
def friendly(n: Int): List[(Int, Int)] =
  def sigma(k: Int) =
    (1 to k).filter(k % _ == 0).sum
  (for
    i ← 1 to n
    j ← i + 1 to n
    if sigma(i) * j == sigma(j) * i
  yield (i, j)).toList
```

Subtyping (14 pts)

Recall that for any two types `T1` and `T2`, `T1 <: T2` means `T1` is a subtype of `T2`.

Recall that `+` means covariance, `-` means contravariance and no annotation means invariance (i.e., neither covariance nor contravariance).

Consider the following type definitions:

```
trait Bldg[-A]:
  def fill(a: A): Unit

trait Food
trait Rest[P] extends Bldg[P]
```

For each of the following code fragments, indicate whether the definition respects variance and subtyping rules: *Yes* if the code is correct, and *No* if variance or subtyping errors would cause it to be rejected by the compiler.

Question 12 Is the following code valid?

```
trait Fact[+P, -E, +W] extends Bldg[W]
```

☐ Yes ■ No

Question 13 Is the following code valid?

```
trait Fact[+P, -E, W] extends Bldg[W]
```

■ Yes ☐ No

Question 14 Is the following code valid?

```
trait Fact[+P, -E, -W] extends Bldg[Bldg[W ⇒ E] ⇒ P]
```

☐ Yes ■ No

Note Here is how to answer this question:

```
Bldg[Bldg[W => E] => P]
    ~~~~~~~~~~~~~~~~~~~ contra (from Bldg)
     ~~~~~~~~~~~~       contra (from =>)
         ~~~~~~~~       contra (from Bldg)
                     ~      co (from =>)
               ~            co (from =>)
          ~             contra (from =>)
```

So `P` is in a contravariant context, `W` is in a covariant context and `E` is in a contravariant context: `Fact` could be declared as `Fact[-P, -E, +W]`.

Question 15 Is the following code valid?

```
def f[T, U <: T](b: Bldg[Int ⇒ Bldg[T]], r: Rest[U]): Unit =
  b.fill(i ⇒ r)
```

☐ Yes ■ No

Consider also the following classes:

```
class Vector[+T]
class Function[-T, +Q]
class Set[T]
```

Question 16 Is it the case that
`Set[Set[Int]] <: Set[Int]` ?

☐ Yes ■ No

Question 17 Is it the case that
`Function[Bldg[Any], Rest[Int]] <: Function[Rest[Int], Rest[Any]]` ?

☐ Yes ■ No

Question 18 Is it the case that
`Vector[Bldg[Int] ⇒ Bldg[Any]] <: Vector[Rest[Any] ⇒ Bldg[Set[Int]]]` ?

■ Yes ☐ No

Parallelism (16 pts)

In this exercise, we will take a look at parallel collections and operations over them. Your task is to reason about the correctness and safety of parallelized operations.

A useful analogue to `foldLeft` is `scanLeft`, which produces a list of intermediate values of the accumulator. Here is a REPL session that exemplifies its behavior:

```
scala> List.empty[Int].scanLeft(0)((x, y) ⇒ x + y)
val res0: List[Int] = List(0)

scala> List(1, 2, 3).scanLeft(0)(_ + _)
val res1: List[Int] = List(0, 1, 3, 6)

scala> List(1, 2, 3).scanLeft(5)(_ + _)
val res2: List[Int] = List(5, 6, 8, 11)

scala> List(1, 2, 3).scanLeft(5)(_ - _)
val res3: List[Int] = List(5, 4, 2, -1)
```

Similarly, `scanRight` generalizes `foldRight` by tracking intermediate results:

```
scala> List.empty[Int].scanRight(0)((x, y) ⇒ x + y)
val res0: List[Int] = List(0)

scala> List(1, 2, 3).scanRight(0)(_ + _)
val res1: List[Int] = List(6, 5, 3, 0)

scala> List(1, 2, 3).scanRight(5)(_ + _)
val res2: List[Int] = List(11, 10, 8, 5)

scala> List(1, 2, 3).scanRight(5)(_ - _)
val res3: List[Int] = List(-3, 4, -2, 5)
```

Signature information and some documentation for `scanLeft` and `scanRight` for a list of type `List[A]` are given below:

```
extension [A](l: List[A])
  /* Produces a collection containing cumulative results of
     applying the operator going left to right, including the
     initial value. */
  def scanLeft[B](z: B)(op: (B, A) ⇒ B): List[B]

  /* Produces a collection containing cumulative results of applying
     the operator going right to left. */
  def scanRight[B](z: B)(op: (A, B) ⇒ B): List[B]
```

Equational reasoning

It is often possible to express a function in terms of other functions. For example, for all `l: List[T]` and `f: T => List[T]`, `l.flatMap(f) === l.map(f).flatten`.

Hence, we may naturally ask: is `scanRight` really necessary, or can all calls of the form `l.scanRight(z)(op)` be rewritten to calls to `scanLeft` with appropriate modifications to the input list `l`, the base value `z`, and the accumulation function `op`? Answer this question by writing down an equality relation between `scanRight` and `scanLeft` valid for all base values `z: B`, all lists `l: List[A]` and all accumulation functions `op: (B, A) => B`, or write `NONE` if no such relation exists:

```
l.scanLeft(z)(op) == l.reverse.scanRight(z)((a, b) ⇒ op(b, a)).reverse
```

Parallelism

`scanLeft` specifies in which order in which the function `op` is applied. Yet, as for `foldLeft`, its output is actually independent of parenthesization choices when the type `A` is the same as `B` and `op` is associative (in that case, `op(op(op(z, a0), a1), a2) === op(z, op(a0, op(a1, a2)))`, for example.

Below are 6 candidate implementations of `scanLeft`, assuming an associative `op`. An implementation is considered *correct* if and only if correctly implements the `scanLeft` specification above, assuming that `op` is associative.

Question 19
Is the implementation `scanLeft1` correct?

```
extension [B](l: List[B])
  def scanLeft1(z: B)(op: (B, B) => B): List[B] =
    l match
      case Nil    => Nil
      case h :: t => t.scanLeft1(op(z, h))(op)
```

☐ Yes ■ No

Question 20
Is the implementation `scanLeft2` correct?

```
extension [B](l: List[B])
  def scanLeft2(z: B)(op: (B, B) => B): List[B] =
    l.par.map(a => op(z, a)).toList
```

☐ Yes ■ No

For the following questions, consider the following definitions:

```
enum ScanTree[B]:
  val b: B

  case SLeaf(b: B)
  case SBranch(b: B, l: ScanTree[B], r: ScanTree[B])

  def reduceLeft[A1, A2](z: A1)(
      leafOp: A1 => A2,
      seqOp: (A1, B) => A1,
      combOp: (A2, A2) => A2
  ): A2 =
    def loop(tr: ScanTree[B], acc: A1): A2 =
      tr match
        case SLeaf(b) => leafOp(seqOp(acc, b))
        case SBranch(_, l, r) =>
          combOp(loop(l, acc), loop(r, seqOp(acc, l.b)))
    loop(this, z)
import ScanTree.*
```

Question 21

Is the implementation `scanLeft3` correct?

```
extension [B](l: List[B])
  def scanLeft3(z: B)(op: (B, B) ⇒ B): List[B] =
    def mkTree0(l: List[ScanTree[B]]): List[ScanTree[B]] =
      l match
        case h1 :: h2 :: tl ⇒
          SBranch(op(h1.b, h2.b), h1, h2) :: mkTree0(tl)
        case _ ⇒ l
    def mkTree(l: List[ScanTree[B]]): ScanTree[B] =
      l match
        case List(tr) ⇒ tr
        case _        ⇒ mkTree(mkTree0(l))
    def reduce(tr: ScanTree[B], acc: B): List[B] =
      tr match
        case SLeaf(b) ⇒ List(op(acc, b))
        case SBranch(_, l, r) ⇒
          reduce(l, acc) ++ reduce(r, op(acc, l.b))
    z :: {
      if l.isEmpty then List()
      else reduce(mkTree(l.map(b ⇒ SLeaf(b))), z)
    }
```

■ Yes ☐ No

Question 22

Is the implementation `scanLeft4` correct?

```
extension [B](l: List[B])
  def scanLeft4(z: B)(op: (B, B) ⇒ B): List[B] =
    l.foldLeft(z :: Nil)((bs, a) ⇒ op(bs.head, a) :: bs)
      .reverse
```

■ Yes ☐ No

Question 23

Is the implementation `scanLeft5` correct?

```
extension [A](l: List[A])
  def scanLeft5[B](z: B)(op: (B, A) ⇒ B): List[B] =
    l.par.aggregate(z)(op, (l1, l2) ⇒ l1 ++ l2)
```

☐ Yes ■ No

Question 24

Is the implementation `scanLeft6` correct? For this question, assume that `reduce` is well defined for non-associative operations, and applies its operator according to an arbitrary parenthesization of the input.

```
extension [B](l: List[B])
  def scanLeft6(z: B)(op: (B, B) ⇒ B): List[B] =
    z :: {
      if l.isEmpty then Nil
      else
        l.par.map(b ⇒ SLeaf(b))
          .reduce((l, r) ⇒ SBranch(op(l.b, r.b), l, r))
          .reduceLeft(z)(b ⇒ List(b), op, _ ++ _)
    }
```

■ Yes ☐ No

Completely balanced trees (15 points)

Question 25 This question is worth 15 points.

☐ 0 ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7
☐ 8 ☐ 9 ☐ 10 ☐ 11 ☐ 12 ☐ 13 ☐ 14 ■ 15

Do not write here.

Consider the following definitions:

```
enum Tree:
  case Empty
  case Branch(left: Tree, right: Tree)
```

The size of a binary tree is defined thus:

```
extension (that: Tree)
  def size: Int =
    that match
      case Empty               ⇒ 0
      case Branch(left, right) ⇒ 1 + left.size + right.size
```

For the purpose of this exercise, a tree is *locally balanced* if it is empty or if it is a `Branch` and both of its subtrees are of sizes diverging by at most one. A tree is *completely balanced* if all of its subtrees are locally balanced. These properties can be checked using the following function:

```
extension (that: Tree)
  def isLocallyBalanced: Boolean = that match
    case Empty               ⇒ true
    case Branch(left, right) ⇒ math.abs(left.size - right.size) ≤ 1

  def isCompletelyBalanced: Boolean =
    that match
      case Empty ⇒ true
      case Branch(left, right) ⇒
        that.isLocallyBalanced &&
        left.isCompletelyBalanced &&
        right.isCompletelyBalanced
```

Your task is to complete a function `completelyBalanced` that constructs all completely balanced binary trees of a given size. The function returns a list of trees; the order of trees in that list does not matter.

You should only write in the boxes on the next page.

```
import Tree.*
def completelyBalanced(size: Int): List[Tree] =
  if size == 0 then
    List(Empty)
  else if size % 2 == 1 then
    val tr = completelyBalanced(size / 2)
    for
      t0 ← tr
      t1 ← tr
    yield Branch(t0, t1)
  else
    val tr = completelyBalanced(size / 2)
    val tr1 = completelyBalanced(size / 2 - 1)
    for
      t ← tr
      t1 ← tr1
      t ← List(Branch(t, t1), Branch(t1, t))
    yield t
```

Appendix: Scala Standard Library Methods

Here are the prototypes of some Scala classes that you might find useful:

```
// Time complexity is listed for some methods below in big-O notation.
// n refers to the number of elements in the list.
abstract class List[+A]:
  // Adds an element at the beginning of this list. O(1)
  def ::[B >: A](elem: B): List[B]
  // Get the element at the specified index. O(n)
  def apply(n: Int): A
  // Tests whether this list contains a given value as an element. O(n)
  def contains[A1 >: A](elem: A1): Boolean
  // Selects all elements except first n ones.
  def drop(n: Int): List[A]
  // Drops longest prefix of elements that satisfy a predicate.
  def dropWhile(p: A => Boolean): List[A]
  // Selects all elements of this list which satisfy a predicate.
  def filter(pred: A => Boolean): List[A]
  // Selects all elements of this list which do not satisfy a predicate.
  def filterNot(pred: A => Boolean): List[A]
  // Builds a new list by applying a function to all elements of this list and
  // using the elements of the resulting collections
  def flatMap[B](f: A => List[B]): List[B]
  // Applies a binary operator to a start value and all elements of this
  // sequence, going left to right.
  def foldLeft[B](z: B)(op: (B, A) => B): B
  // Applies a binary operator to a start value and all elements of this
  // sequence, going right to left.
  def foldRight[B](z: B)(op: (A, B) => B): B
  // Tests whether a predicate holds for every element of this collection
  def forall(p: A => Boolean): Boolean
  // Selects the first element of this list. O(1)
  def head: A
  // Computes the multiset intersection between this sequence and another sequence.
  // O(n*m), where m is the number of elements in 'that'
  def intersect[B >: A](that: Seq[B]): List[A]
  // Selects the last element. O(n)
  def last: A
  // Applies the function f to each element in the list.
  def map[B](f: A => B): List[B]
  // Returns a new list with elements in reversed order. O(n)
  def reverse: List[A]
  // The size of this collection. O(n)
  def size: Int
  // Sorts this sequence according to an Ordering. O(n * log(n))
  def sorted[B >: A](implicit ord: Ordering[B]): List[A]
  // Selects all elements except the first. O(1)
  def tail: List[A]
  // Takes longest prefix of elements that satisfy a predicate.
  def takeWhile(p: A => Boolean): List[A]

object List:
  // Produces a collection containing the results of some element computation a
  // number of times.
  def fill[A](n: Int)(elem: => A): List[A] = ???
```

Solutions

```
abstract class ParList[+A] extends List[A]:
  // Aggregates the results of applying an operator to subsequent elements.
  def aggregate[B](z: ⇒ B)(seqop: (B, A) ⇒ B, combop: (B, B) ⇒ B): B

abstract class Option[+A]:
  // Returns this option's value.
  def get: A
  // Returns true if this option is an instance of Some, false otherwise.
  def isDefined: Boolean
  // Returns true if this option is None, false otherwise.
  def isEmpty: Boolean

object math:
  // Returns the value rounded down to an integer.
  def floor(x: Double): Double = ???
  // Returns the value of the first argument raised to the power of the second
    argument.
  def pow(x: Double, y: Double): Double = ???
  // Returns the square root of a Double value.
  def sqrt(x: Double): Double = ???

abstract class Double:
  // Converts this value to an integer
  def toInt: Int
```