Corrected

EPFL

Profs. Viktor Kunčak, Martin Odersky, and Clément Pit-Claudel

CS-214 Software Construction Midterm
Nov 5, 2025, 16:15–18:15
Duration: 120 minutes

1

2025

SCIPER : **1000001** Room : **BCH 2201** Signature :

Wait for the start of the exam before turning to the next page. This booklet is printed double-sided, 24 pages. Do not unstaple or detach any pages.

Materials	This is a closed-book exam. Paper documents and electronic devices are not allowed. Place your student ID and writing utensils on your desk. 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: exercises that have the same number of points may not be equally difficult. Look at the whole exam before starting.
Appendix	The last pages of this exam contain an appendix with useful function signatures. You may use functions from the Scala standard library even if they do not appear in the appendix.
Use a pen	To avoid scanning errors, **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 75 points. The points for each question are indicated next to it.
Stay functional and clean	Do not use `var`s, `while` loops, `for...do` loops, etc. (this will result in 0 points for that question). Points are attributed for *correctness*, *succinctness*, *efficiency*, and *simplicity*.

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

1 Refactoring (6 pts)

Some or all the following snippets of code can be refactored using the `List` API. **Select which list function** is the **optimal** choice, if any. Then **write** the refactored version in the box. If none of the proposed choices work, check "None" and leave the box blank.
Choose only one option.

Question 1 *(2 points)*

☐ 0 ☐ 1 ■ 2 *Do not write here.*

```
def example1(l: List[Int]): List[Int] =
  l match
    case Nil => Nil
    case head :: tail =>
      head + 1 :: example1(tail)
```

```
def example1(l: List[Int]): List[Int] =
  l.map(_ + 1)
```

☐ foldLeft ☐ foldRight ☐ reduceLeft ☐ reduceRight ■ map
☐ forall ☐ filter ☐ flatMap ☐ None

Question 2 *(2 points)*

☐ 0 ☐ 1 ■ 2 *Do not write here.*

```
def example2(l: List[Int]): List[Int] =
  l match
    case Nil => Nil
    case head :: tail =>
      if head % 2 == 0
      then head :: example2(tail)
      else example2(tail)
```

```
def example2(l: List[Int]): List[Int] =
  l.filter(_ % 2 == 0)
```

☐ foldLeft ☐ foldRight ☐ reduceLeft ☐ reduceRight ☐ map
☐ forall ■ filter ☐ flatMap ☐ None

Question 3 *(2 points)*

☐ 0 ☐ 1 ■ 2 *Do not write here.*

```
def example3(l: List[Int]): Boolean =
  l match
    case Nil => true
    case head :: tail =>
      if head > 10
      then true && example3(tail)
      else false && example3(tail)
```

```
def example3(l: List[Int]): Boolean =
  l.forall(_ > 10)
```

☐ foldLeft ☐ foldRight ☐ reduceLeft ☐ reduceRight ☐ map
■ forall ☐ filter ☐ flatMap ☐ None

2 Proof (8 pts)

All lemmas on this page hold for all types `T` and `S`; `f` of type `T => S`; `x` and `y` of type `T`; and `lst1`, `lst2`, `xs`, `ys`, `zs` of type `List[T]`.

Your task is to prove MapAppend:

(MapAppend) `(lst1 ++ lst2).map(f) === lst1.map(f) ++ lst2.map(f)`

The proof is done by induction on `lst1`. You are given the following lemmas:

(NilAppend) `Nil ++ xs === xs`

(AppendNil) `xs ++ Nil === xs`

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

(AppendCons) `xs ++ (y :: ys) === (xs ++ (y :: Nil)) ++ ys`

(AppendAssoc) `xs ++ (ys ++ zs) === (xs ++ ys) ++ zs`

(MapNil) `Nil.map(f) === Nil`

(MapCons) `(x :: xs).map(f) === f(x) :: xs.map(f)`

Complete each case of the proof below by writing a sequence of equalities. Write every step of reasoning. For each step, you may only use one of the lemmas above or the induction hypothesis in the inductive case, and you must write the name of the lemma you are using. You do not need to use all of the given lemmas:

Question 4 *(3 points)*

☐0 ☐1 ☐2 ■3 *Do not write here.*

Base case: `lst1` is `Nil`. Therefore, you need to prove:

`(Nil ++ lst2).map(f) === Nil.map(f) ++ lst2.map(f)`

```
    (Nil ++ lst2).map(f)
=== lst2.map(f)                  by NilAppend
=== Nil ++ lst2.map(f)           by NilAppend
=== Nil.map(f) ++ lst2.map(f)    by MapNil
```

Corrected

Question 5 *(5 points)*

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

Inductive case: `lst1` is `x :: xs`. Therefore, you need to prove:

```
((x :: xs) ++ lst2).map(f) === (x :: xs).map(f) ++ lst2.map(f)
```

given that the induction hypothesis, named IH, is true: for all `ys` of type `List[T]`,

```
(xs ++ ys).map(f) === xs.map(f) ++ ys.map(f)
```

```
    ((x :: xs) ++ lst2).map(f)
=== (x :: (xs ++ lst2)).map(f)          by ConsAppend
=== f(x) :: (xs ++ lst2).map(f)         by MapCons
=== f(x) :: (xs.map(f) ++ lst2.map(f))  by IH
=== (f(x) :: xs.map(f)) ++ lst2.map(f)  by ConsAppend
=== (x :: xs).map(f) ++ lst2.map(f)     By MapCons
```

3 Tree symmetry (18 pts)

In this exercise, you will work with binary trees, focusing on *mirroring* and *symmetry*.

Here is the structure that you will use throughout the exercise:

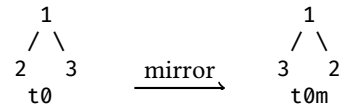
```
case class Tree[T](value: T, left: Option[Tree[T]], right: Option[Tree[T]])
def Leaf[T](value: T): Some[Tree[T]] = Some(Tree(value, None, None))
```

3.1 Mirroring trees (4 pts)

3.1.1 Definition

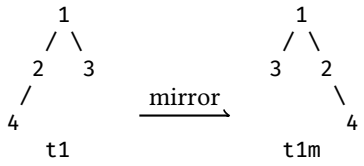
Given a two-dimensional picture, we can define its "vertical mirror image" by flipping it along a vertical axis, as in the example to the right.
We can perform a similar transformation on binary trees, producing a "vertical mirror tree". Below are two examples:

Example 0:




```
val t0 = Tree(1, Leaf(2), Leaf(3))
val t0m = Tree(1, Leaf(3), Leaf(2))
assertEquals(mirror(t0), t0m)
```

Example 1:




```
val t1 = Tree(1, Some(Tree(2, Leaf(4), None)), Leaf(3))
val t1m = Tree(1, Leaf(3), Some(Tree(2, None, Leaf(4))))
assertEquals(mirror(t1), t1m)
```

Question 6 *(4 points)*

0 1 2 3 4 *Do not write here.*

Implement the function `mirror`, which returns the vertical mirror of its input tree. Be succinct.

```
def mirror[T](node: Tree[T]): Tree[T] =
  node.copy(
    left = node.right.map(mirror),
    right = node.left.map(mirror)
  )

// Less succinct!
def otherMirror[T](node: Tree[T]): Tree[T] =
  (node.left, node.right) match
    case (None, None) => node
    case (Some(l), Some(r)) =>
      Tree(node.value, Some(otherMirror(r)), Some(otherMirror(l)))
    case (Some(l), None) =>
      Tree(node.value, None, Some(otherMirror(l)))
    case (None, Some(r)) =>
      Tree(node.value, Some(otherMirror(r)), None)
```

3.2 Tree symmetry (14 pts)

3.2.1 Definition

A two-dimension drawing can have "vertical symmetry". For example, the three pictures on the left are vertically symmetric, whereas the three pictures on the right are not:

We can define a similar notion on trees. The three trees on the left are symmetric, whereas the three trees on the right are not:

```
    1                    true          "hello"
   / \                  /    \        /     \
  2   2              false   false   "hi"   "hi"
 /     \
3       3
```

```
      1                    true          "hello"
     / \                  /    \        /     \
    2   2              false   true    "hi"   "ih"
   /
  3
```

You are given the following definitions:

- The *children* of a tree are the **non-`None`** subtrees among `left` and `right`. Each tree has 0, 1, or 2 children.
- The *descendants* of a tree are its children, plus their children, and so on, recursively.

You are also given the following functions:

```
// Returns the size of a tree
def size[T](tree: Tree[T]): Int =
  val l = tree.left.map(size).getOrElse(0)
  val r = tree.right.map(size).getOrElse(0)
  1 + l + r

// Returns the in-order traversal of a tree
def inOrder[T](tree: Tree[T]): Seq[T] =
  val l = tree.left.map(inOrder).getOrElse(Seq.empty)
  val r = tree.right.map(inOrder).getOrElse(Seq.empty)
  (l :+ tree.value) ++ r

// Returns the set of elements in the tree
def elements[T](tree: Tree[T]): Set[T] =
  inOrder(tree).toSet
```

3.2.2 Part 1: Counterexamples

The following definition of a symmetric tree is **wrong**:

> "*A tree is symmetric if and only the following property is true for it and all of its descendants: 'I have no children, or all of my children are of the same size and contain the same set of elements'.*"

The following counterexamples show that this definition is incorrect:

Counterexample 0.1: false negative
(symmetric tree, incorrectly rejected)

```
          1
        /   \
       2     2
      / \   / \
     3   2 2   3
    /           \
   2             2
```

Counterexample 0.2: false positive
(non-symmetric tree, incorrectly accepted)

```
          1
        / \
       2   3
      /     \
     3       2
    / \     / \
   0   0   0   0
```

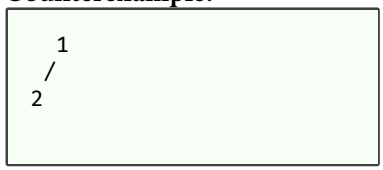
Question 7 *(1 point)*

☐$_0$ ■$_1$ *Do not write here.*

Counterexample 0.2 above is not *minimal*: there are smaller trees that show that definition 0 is wrong.

Draw a *minimal false positive* (a minimal non-symmetric tree that this definition accepts) in the box to the right.

Counterexample:



Question 8 *(2 points)*

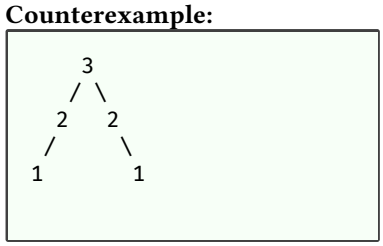
☐$_0$ ☐$_1$ ■$_2$ *Do not write here.*

The following definition of a symmetric tree is **wrong**:

> "*A tree is symmetric if and only if its children are all equal.*"

Draw a *minimal false negative* (a minimal symmetric tree that this definition does not accept) in the box to the right.

Counterexample:



Question 9 *(2 points)*

☐$_0$ ☐$_1$ ■$_2$ *Do not write here.*

The following definition of a symmetric tree is **wrong**:

> "*A tree is symmetric if and only if it has no children or all of its children are symmetric.*"

Draw a *minimal false positive* (a minimal non-symmetric tree that this definition accepts) in the box to the right.

Counterexample:

Question 10 *(2 points)*

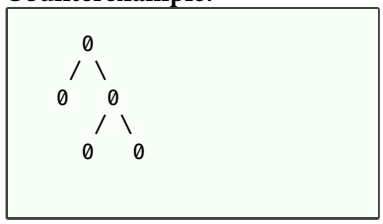
☐$_0$ ☐$_1$ ■$_2$ *Do not write here.*

The following definition of a symmetric tree is **wrong**:

> "*A tree is symmetric if and only if it has no children, or if it has two children and both of them are symmetric.*"

Draw a *minimal false positive* (a minimal non-symmetric tree that this definition accepts) in the box to the right, **using only the value `0`** (repeated as many time as needed).

Counterexample:



Question 11 *(2 points)*

☐₀ ☐₁ ■₂ *Do not write here.*

The following definition may be correct or incorrect:

> "*A tree is symmetric if and only if it is equal to its vertical mirror (as defined in the previous exercise).*"

Indicate whether it is correct or not. If it is incorrect, **draw** a *minimal* counterexample in the box to the right.

■ Correct ☐ Incorrect

Counterexample:

Question 12 *(2 points)*

☐₀ ☐₁ ■₂ *Do not write here.*

The following definition may be correct or incorrect:

> "*A tree is symmetric if and only if an in-order traversal produces a palindromic sequence (i.e. a sequence that reads the same forwards and backward, for example "4, 5, 1, 5, 4").*"

Indicate whether it is correct or not. If it is incorrect, **draw** a *minimal* counterexample in the box to the right.

☐ Correct ■ Incorrect

Counterexample:

3.2.3 Part 2: Executable specifications

Question 13 *(3 points)*

☐₀ ☐₁ ☐₂ ■₃ *Do not write here.*

Write the function `isSymmetric` so that it returns `true` if and only if the input `tree: Tree[T]` is a symmetric tree. You may use the functions defined in the previous part. Be succinct.

```
def isSymmetric[T](tree: Tree[T]): Boolean =
  mirror(tree) == tree
```

4 Variance (5 pts)

Question 14 *(2 points)*

The following code snippet has a variance error, and will be rejected by the Scala compiler:

```
4  trait I[+T]:
5    def t: T
6    def f(t: T): T
```

Select the error message that best describes the problem:

☐
```
-- Error: src/q.scala:5:8 ----------------------------------------------------------
5 |    def t: T
  |    ^^^^^^^^
  |    contravariant type T occurs in covariant position in type => T of method t
```

☐
```
-- Error: src/q.scala:6:8 ----------------------------------------------------------
6 |    def f(t: T): T
  |    ^^^^^^^^^^^^^
  |    contravariant type T occurs in covariant position in type (t: T): T of method f
```

■
```
-- Error: src/q.scala:6:10 ---------------------------------------------------------
6 |    def f(t: T): T
  |          ^^^^
  |          covariant type T occurs in contravariant position in type T of parameter t
```

Question 15 *(3 points)*

☐0 ☐1 ☐2 ■3 *Do not write here.*

Construct a small program which would fail (throw an exception or crash) at runtime if the Scala compiler were to accept the trait definition above. In other words, show a program that would go wrong if variance checks were disabled for the trait definition above. **Keep your solution succinct.**

Your answer may only use usual constructions and applications of Scala functions and values; the types `Int`, `String`, `Boolean`, and `Any`, and their methods; and new classes defined by you that implement the trait `I`.

```
class C extends I[Int]:
  def t: Int = 0
  def f(a: Int): Int = a + 1

val cInt: I[Int] = C()
val cAny: I[Any] = cInt  // OK because I is covariant

val result = cAny.f("Hello") // Boom
```

5 Parallelism and complexity (14 pts)

In this section, we will use the `.par` method on collections to perform computations in parallel (this method becomes available after importing `scala.collection.parallel.CollectionConverters.*`).

5.1 Word statistics (6 pts)

In this exercise, you will compute statistics for very large sequences of words. To improve performance, we use the parallel `aggregate` function. This function first folds across contiguous subsequences of the input using `seqop`, then combines the results with `combop`.

```
abstract class ParSeq[+A] extends Seq[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
```

You are asked to complete the invocations by providing the body of `seq` or `comb`. **Keep your code succinct.**

Example task: Count the number of words in a sentence.
Solution:

```
def countWords(words: Seq[String]): Int =
  def seq(count: Int, word: String) = count + 1
  def comb(c1: Int, c2: Int) = c1 + c2
  words.par.aggregate(0)(seq, comb)
```

Question 16 *(2 points)*

Task: Count how many words are considered "special" based on the given predicate. For example, the following test should pass:

```
test("wordstats: countSpecial"):
  val s = List("I", "love", "Scala", "so", "much")
  assertEquals(countSpecial(s, _.length == 4), 2)
  assertEquals(countSpecial(s, _.startsWith("t")), 0)
```

Complete the `seq` function below:

```
def countSpecial(words: Seq[String], isSpecial: String => Boolean): Int =
  def seq (count: Int, word: String) =
    count + (if isSpecial(word) then 1 else 0)
  def comb (c1: Int, c2: Int) = c1 + c2
  words.par.aggregate(0)(seq, comb)
```

Question 17 *(4 points)*

☐0 ☐1 ☐2 ☐3 ■4 *Do not write here.*

Task: Compute the frequency of each word, ignoring case differences. For example, the following test should pass:

```
test("wordstats: freqs"):
  val s = List("To", "be", "or", "not", "to", "be")
  assertEquals(freqs(s), Map("to" -> 2, "be" -> 2, "or" -> 1, "not" -> 1))
```

Complete the `comb` function below.

```
type Frequencies = Map[String, Int]
def freqs(words: Seq[String]): Frequencies =
  def seq (acc: Frequencies, word: String) =
    acc.updated(word, acc.getOrElse(word, 0) + 1)
  def comb (freqs1: Frequencies, freqs2: Frequencies) =
    freqs2.foldLeft(freqs1)((acc, freq) =>
      val (word, count) = freq
      acc.updated(word, acc.getOrElse(word, 0) + count)
    )
  words.par.map(_.toLowerCase(Locale.ROOT)).aggregate(Map.empty)(seq, comb)
```

5.2 Complexity (8 pts)

Your are given the following definition of the `Boid` class and functions.

```
// You can assume that `Vector2(x:Float, y:Float)` supports addition to and
// subtraction from a `Vector2`, multiplication and division by a `Float`, and
// the `norm: Float` method.  Calling `.sum` on a sequence of `Vector2` works
// as expected, using the `+` method.
case class Boid(position: Vector2, velocity: Vector2)

// Compute the avoidance force from other boids to this boid
def avoidanceForcePar(thisBoid: Boid, otherBoids: Vector[Boid]): Vector2 =
  otherBoids.par                                          // [1]
    .map(b => thisBoid.position + b.position)
    .map(diff => diff / 2)
    .sum

// Compute the cohesion force from other boids to this boid
def cohesionForcePar(thisBoid: Boid, otherBoids: Vector[Boid]): Vector2 =
  otherBoids.par.map(_.position).sum                      // [2]
    / (otherBoids.par.length.toFloat + 1)                 // [3]
    - thisBoid.position

// Tick a boid, compute its next position and velocity using cohesion and avoidance forces
def tickBoidPar(thisBoid: Boid, allBoids: Vector[Boid]): Boid =
  val acc = avoidanceForcePar(thisBoid, allBoids) +
    cohesionForcePar(thisBoid, allBoids)
  Boid(
    thisBoid.position + thisBoid.velocity,
    thisBoid.velocity + acc
  )

// Tick all boids
def tickWorld(allBoids: Vector[Boid]): Vector[Boid] =
  allBoids.par.map(boid => tickBoidPar(boid, allBoids)).seq     // [4]
```

Question 18 *(3 points)*

☐0 ☐1 ☐2 ■3 *Do not write here.*

What are the work and depth complexity of `tickWorld`? Express them as a function of the size n of `allBoids`.

- $W(n) = O(n^2)$: for n boids, we compute $2n$ forces.
- $D(n) = O(\log(n))$: `tickWorld` is completely parallelizable, but for each boid we must then compute both forces, each of which requires time $O(\log(n))$ due to the `sum`.

Rubric: -2 points per mistake.

Question 19 *(2 points)*

☐0 ☐1 ■2 *Do not write here.*

What would be the work and depth complexity of the `tickWorld` function if all `.par` were removed? Express them as a function of the size n of `allBoids`.

- $W(n) = O(n^2)$: work complexity won't change depending on parallelism changes.
- $D(n) = O(n^2)$: as we don't have any parallelism, the depth is now the same as the work.

Rubric: 1 point for unchanged work; 1 point for equal work and depth.

Corrected

Question 20 *(3 points)*

A senior developer looked at this code and said:

> "Those `par`s! They hurt my eyes! Get rid of as many of them as possible without impacting performance too badly for our average customer."

The average customer has a classic computer with 10 threads and runs the simulation with 1000 boids.
Check as many `.par` as possible that can be removed together without badly impacting performance.

- [x] `Remove .par number 1 (in avoidanceForce)`
- [x] `Remove .par number 2 (in cohesion force, before .map)`
- [x] `Remove .par number 3 (in cohesion force, before .length)`
- [ ] `Remove .par number 4 (in tickWorld)`

`[1]` and `[2]` are superfluous because all available parallelism will already be exhausted by `[4]`; `[3]` is completely useless. Removing just `[3]` and `[4]` may perform a bit better in some circumstances, but the question asked to remove as many `.par`s as possible.

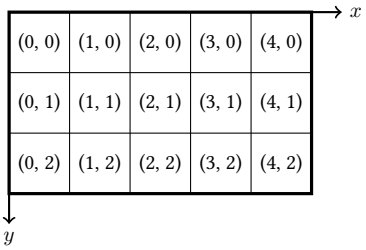
6 Campus renovation (23 pts)

EPFL is planning a major renovation of its campus. The goal is to maximize available classroom seating capacity while keeping a tight budget.

6.1 Definitions

We model the problem as follows:

- A campus plan is a two-dimensional grid of size `(xSize, ySize)` with a collection of buildings.
- A coordinate `(x, y)` defines a specific cell in the grid. Both `x` and `y` must be non-negative.

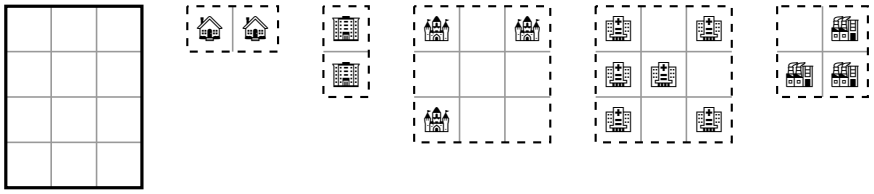


A building is described by:

- `id`: its identifier, for example `"single deck"`;
- `coords`: the cells that it occupies in the campus, stored as a (non-empty) set of coordinates;
- `price`: its (non-negative) price in Swiss Francs;
- `seats`: the total number of seats provided by its classrooms.

In the following, we use the words “campus”, “plan” and “campus plan” interchangeably.

In the diagram below, we illustrate an empty campus of size (3, 4) along with five buildings of various shapes and sizes.



Buildings 🏠 and 🏢 each occupy two coordinates; building 🏰 and 🏭 each occupy three coordinates; building 🏥 occupies six coordinates.

Note that some buildings are disconnected (🏰 is one such example).

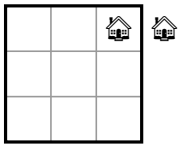
6.2 Problem statement

The aim of this exercise is to find a *feasible* campus plan that maximizes seating capacity.

- A plan is *feasible* when it is *affordable* and *valid*.
  - A plan is *affordable* if the total cost of the buildings is at most a given budget.
  - A plan is *valid* if it has no *overlapping* buildings and no buildings that extend *out of campus bounds*.
    - Two buildings *overlap* if they have at least one coordinate in common.
    - A building *extends out of the campus* if one of its coordinates is not within campus boundaries.

Here are some examples:

Building 🏠 extends out of campus bounds



Buildings 🏠 and 🏢 overlap (at (2, 0))

Valid plan

Valid plan

We define the following classes to reason about the campus renovation. You can assume that the methods defined here are correct and you are free to use them.

```
case class Coord(x: Int, y: Int):
  require(x >= 0 && y >= 0)
  // Component-wise addition: (1,2) + (3,4) = (4,6)
  def +(other: Coord): Coord = Coord(x + other.x, y + other.y)
```

```
case class CampusPlan(buildings: List[Building], xSize: Int, ySize: Int):
  // Return the same campus with the given building added
  def :+(b: Building): CampusPlan =
    require(b.fits(this))
    copy(buildings = b :: buildings)
```

```
case class Building(id: String, coords: Set[Coord], price: Int, seats: Int):
  // Checks that this building does not extend out of bounds of the campus
  def inBoundaries(c: CampusPlan): Boolean = // ...
```

Keep your code short and simple.

6.3 Fitting and translating (8 pts)

Question 21 *(4 points)*

Do not write here.

A building `b` *fits* within a valid campus `c` if `c :+ b` is valid; that is, if the campus remains valid after adding the building to it. Write an extension method `fits` of `Building` such that `b.fits(c)` returns whether `b` fits in `c`. You may assume that `c` is valid.

```
extension (b: Building)
  def fits(c: CampusPlan): Boolean =
    b.inBoundaries(c) &&
      b.coords.intersect(c.buildings.toSet.flatMap(_.coords)).isEmpty
  def fits2(c: CampusPlan): Boolean =
    b.inBoundaries(c) &&
      c.buildings.forall(_.coords.intersect(b.coords).isEmpty)
  def fits3(c: CampusPlan): Boolean =
    b.inBoundaries(c) &&
      c.buildings.forall(b2 =>
        !b.coords.exists(b2.coords.contains))
```

For the next two methods, we define `Offset` as:

```
type Offset = Coord
```

Question 22 *(2 points)*

☐0 ☐.5 ☐1 ☐.5 ■2 *Do not write here.*

Write an extension method `translated` of `Building` such that `b.translated(d)` returns a copy of `b`, moved by `d` towards the bottom right.

```
extension (b: Building)
  def translated(d: Offset): Building =
    b.copy(coords = b.coords.map(_ + d))
```

Question 23 *(2 points)*

☐0 ☐.5 ☐1 ☐.5 ■2 *Do not write here.*

Write an extension method `fitsWhenTranslated` of `Building` such that `b.fitsWhenTranslated(d)(c)` returns `true` if and only if `b` *fits* in campus `c` when translated by `d`, as defined in question 21.

```
extension (b: Building)
  def fitsWhenTranslated(d: Offset)(c: CampusPlan): Boolean =
    b.translated(d).fits(c)
```

6.4 Find the best plan (15 pts)

Question 24 *(3 points)*

☐0 ☐.5 ☐1 ☐.5 ☐2 ☐.5 ■3 *Do not write here.*

Write the function `allPositions`, which, given a campus dimensions (`xSize` and `ySize`), returns a set of all its coordinates, as defined in the "Definitions" section, at the beginning of the exercise.

```
def allPositions(xSize: Int, ySize: Int): Set[Coord] =
  (for x <- 0 until xSize
       y <- 0 until ySize
  yield Coord(x, y)).toSet
```

Question 25 *(9 points)*

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

Complete the function `allPlans` on the next page, which, given a list of buildings `proposals`, a `campus` and a `budget`, returns the set of all *feasible* plans that can be created from the initial campus plan, using the buildings in `proposals`. Each building may be used at most once in each plan.

You may assume that the initial campus plan is *valid* and does not contain any of the buildings in `proposals`.

Question 26 *(3 points)*

☐ 0 ☐ .5 ☐ 1 ☐ .5 ☐ 2 ☐ .5 ■ 3 *Do not write here.*

Finally, complete the function `bestPlan` so that it returns the *best* plan, i.e. the feasible plan that maximizes the total number of seats in its buildings. If there are multiple plans with the same maximum capacity, you may return any one of them (there is no need to minimize the budget).

You may assume that the initial campus plan is *valid*.

```
def bestPlan(proposals: List[Building], campus: CampusPlan, budget: Int): CampusPlan =
  require(budget >= 0)
  val plans = allPlans(proposals, campus, budget)

  assert(plans.nonEmpty) // This is not mandatory but should hold
  plans.maxBy(_.buildings.map(_.seats).sum)
```

```
// Computes all feasible plans
def allPlans(proposals: List[Building], campus: CampusPlan, budget: Int): Set[CampusPlan] =
  require(budget >= 0)
  if budget == 0 then
    Set(campus)
  else proposals match
    case Nil =>
      Set(campus)
    case b :: rest =>
      val whenNotUsed: Set[CampusPlan] = // Feasible plans that do not use b
        allPlans(rest, campus, budget)

      val whenUsed = // Feasible plans that do use b
        for
          c <- allPositions(campus.xSize, campus.ySize)
          positioned = b.translated(c)
          if positioned.fits(campus)
          if positioned.price <= budget
          plan <- allPlans(rest, campus :+ positioned, budget - b.price)
        yield plan

      // Combine to get all feasible plans
      whenUsed ++ whenNotUsed
```

7 Putting it all together (1 pt)

Question 27 *(1 point)*

☐ 0 ■ 1 *Do not write here.*

CS-214 has two parts, functional programming and software engineering. These two parts share three main themes. Name the three main themes of CS-214:

- Modularity
- Correctness
- Real-world engineering

OR, you may instead solve the following (very difficult) question. There are no bonus points for answering both questions; if you answer both, only the first one (3 themes, above) will be graded. Do not attempt the question below unless you have completed the rest of this exam (except the question above).

Self-similarity 🔥 🔥 🔥:
Write a purely functional program that takes no input and prints its own source code.
Do not use system functions (e.g., you may not read from a file).

This exercise will be scanned and auto-graded. Work on draft paper, and only copy your answer once you're done.

Indicate line breaks with ⏎ and spaces with an empty box. Without any ⏎, we will assume your program is on a single line.

Write legibly, one character per box.

```
val q = "\"" * 3
val c = """val q = "\"" * 3
val c = %s%s%s
println(c format (q, c, q))
"""
println(c format (q, c, q))
```

```
val data = """println(s"val data = \"\"\"$data\"\"\"\n$data")"""
println(s"val data = \"\"\"$data\"\"\"\n$data")
```

```
val x = "val x = %c%s%1$c; printf(x, 34, x)"; printf(x, 34, x)
```

Appendix

```
abstract class List[+A]:
  // Returns a new sequence containing the elements from the left hand operand
  // followed by the elements from the right hand operand.
  def ++[B >: A](suffix: IterableOnce[B]): List[B]
  // A copy of the list with an element prepended.
  def +:[B >: A](elem: B): List[B]
  // A copy of this sequence with an element appended.
  def :+[B >: A](elem: B): List[B]
  // Adds an element at the beginning of this list.
  def ::[B >: A](elem: B): List[B]
  // Selects all the elements of this sequence ignoring the duplicates.
  def distinct: List[A]
  // Selects all elements except the first n ones
  def drop(n: Int): List[A]
  // Tests whether a predicate holds for at least one element of this list.
  def exists(p: A => Boolean): Boolean
  // Selects all elements of this list which satisfy a predicate.
  def filter(p: A => Boolean): List[A]
  // Finds the first element of the list satisfying a predicate, if any.
  def find(p: A => Boolean): Option[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 => IterableOnce[B]): List[B]
  // Applies the given binary operator op to the given initial value z and all
  // elements of this sequence, going left to right. Returns the initial value
  // if this sequence is empty.
  def foldLeft[B](z: B)(op: (B, A) => B): B
  // Applies the given binary operator op to all elements of this list and the
  // given initial value z, going right to left. Returns the initial value if
  // this list is empty.
  def foldRight[B](z: B)(op: (A, B) => B): B
  // Tests whether a predicate holds for all elements of this list.
  def forall(p: A => Boolean): Boolean
  // Partitions this iterable collection into a map of iterable collections
  // according to some discriminator function.
  def groupBy[K](f: A => K): Map[K, List[A]]
  // The initial part of the collection without its last element.
  def init: List[A]
  // Iterates over the inits of this iterable collection.
  def inits: Iterator[List[A]]
  // Selects the last element.
  def last: A
  // Optionally selects the last element.
  def lastOption: Option[A]
  // Builds a new list by applying a function to all elements of this list.
  def map[B](f: A => B): List[B]
  // Applies the given binary operator op to all elements of this collection.
  def reduce[B >: A](op: (B, B) => B): B
  // Applies the given binary operator op to all elements of this collection,
  // going left to right.
  def reduceLeft[B >: A](op: (B, A) => B): B
```

Corrected

```
  // Applies the given binary operator op to all elements of this collection,
  // going right to left.
  def reduceRight[B >: A](op: (A, B) => B): B

  // Returns a new list with the elements of this list in reverse order.
  def reverse: List[A]

  // Selects an interval of elements.
  def slice(from: Int, until: Int): List[A]

  // Sorts this sequence according to a comparison function.
  def sortWith(lt: (A, A) => Boolean): List[A]

  // Sums the elements of this list.
  // You do not need to provide the num parameter explicitly.
  def sum[B >: Int](implicit num: Numeric[B]): B

  // Iterates over the tails of this sequence.
  def tails: Iterator[List[A]]

  // Selects the first n elements.
  def take(n: Int): List[A]

  // Converts this collection to a Set.
  def toSet[B >: A]: Set[B]

  // Returns a iterable collection formed from this iterable collection and
  // another iterable collection by combining corresponding elements in pairs.
  def zip[B](that: IterableOnce[B]): List[(A, B)]

  // ...
```

```
abstract class Map[K, +V]:

  // Creates a new map obtained by updating this map with a given key/value pair.
  def +[V1 >: V](kv: (K, V1)): Map[K, V1]
  def updated[V1 >: V](key: K, value: V1): Map[K, V1]

  // Removes a key from this map, returning a new map.
  def -(key: K): Map[K, V]

  // Tests whether this map contains a binding for a key.
  def contains(key: K): Boolean

  // Applies the given binary operator op to the given initial value z
  // and all elements of this collection, going left to right.
  def foldLeft[B](z: B)(op: (B, (K, V)) => B): B

  // Optionally returns the value associated with a key.
  def get(key: K): Option[V]

  // Returns the value associated with a key, or a default value if the key is
  // not contained in the map.
  def getOrElse[V1 >: V](key: K, default: => V1): V1

  // Builds a new iterable collection by applying a function to all elements of
  // this iterable collection.
  def map[B](f: ((K, V)) => B): Iterable[B]

  // Builds a new map by applying a function to all elements of this map.
  def map[K2, V2](f: ((K, V)) => (K2, V2)): Map[K2, V2]

  // Converts this collection to a List.
  def toList: List[(K, V)]

  // ...
```

```
// Other functions

// Computes `a` and `b` in parallel and returns their results as a tuple
def parallel[A, B](a: => A, b: => B): (A, B) = ???

extension (that: Int)
  // An immutable range from `that` up to but not including `end`
  def until(end: Int): Range = ???
  // An immutable range from `that` up to and including `end`
  def to(end: Int): Range = ???
```

```
// Vector has the same methods as `List`, with the input and output types
// appropriately changed from `List[_]` to `Vector[_]`.
abstract class Vector[+A]
```

```
// Seq has the same methods as `List`, with the input and output types
// appropriately changed from `List[_]` to `Seq[_]`.
abstract class Seq[+A]
```

```
abstract class Set[A]:
  // Creates a new set with an additional element, unless the element is already present.
  def +(elem: A): Set[A]
  // Returns a new iterable collection containing the elements from the left
  // hand operand followed by the elements from the right hand operand.
  def ++[B >: A](suffix: IterableOnce[B]): Set[B]
  // Creates a new set with a given element removed from this set.
  def -(elem: A): Set[A]
  // Creates a new immutable set from this immutable set by removing all
  // elements of another collection.
  def --(that: IterableOnce[A]): Set[A]
  // Computes the difference of this set and another set.
  def diff(that: Set[A]): Set[A]
  // Tests whether a predicate holds for at least one element of this collection.
  def exists(p: A => Boolean): Boolean
  // Selects all elements of this iterable collection which satisfy a predicate.
  def filter(pred: A => Boolean): Set[A]
  // Builds a new iterable collection by applying a function to all elements of
  // this iterable collection and using the elements of the resulting collections.
  def flatMap[B](f: A => IterableOnce[B]): Set[B]
  // Tests whether a predicate holds for all elements of this collection.
  def forall(p: A => Boolean): Boolean
  // Partitions this iterable collection into a map of iterable collections
  // according to some discriminator function.
  def groupBy[K](f: A => K): Map[K, Set[A]]
  // Computes the intersection between this set and another set.
  def intersect(that: Set[A]): Set[A]
  // Returns true if this set is empty
  def isEmpty: Boolean
```

```
  // Returns true if this set contains at least one element
  def nonEmpty: Boolean

  // Builds a new iterable collection by applying a function to all elements of
  // this iterable collection.
  def map[B](f: A => B): Set[B]

  // Finds the maximum element of this set as measured by the given function.
  // You do not need to provide the ord parameter explicitly.
  def maxBy[B](f: A => B)(implicit ord: Ordering[B]): A

  // Applies the given binary operator op to all elements of this collection.
  def reduce[B >: A](op: (B, B) => B): B

  // Tests whether this set is a subset of another set.
  def subsetOf(that: Set[A]): Boolean

  // An iterator over all subsets of this set.
  def subsets(): Iterator[Set[A]]

  // Converts this collection to a List.
  def toList: List[A]

  // ...
```

```
abstract class String:
  // Returns a new string containing the chars from this string followed by the
  // chars from the right hand operand.
  def +(suffix: IterableOnce[Char]): String

  // The rest of the string without its n first chars.
  def drop(n: Int): String

  // Tests if this string ends with the specified suffix.
  def endsWith(suffix: String): Boolean

  // Selects all chars of this string which satisfy a predicate.
  def filter(pred: Char => Boolean): String

  // Builds a new string by applying a function to all chars of this string and
  // using the elements of the resulting strings.
  def flatMap(f: Char => String): String

  // Tests whether a predicate holds for all chars of this string.
  def forall(p: Char => Boolean): Boolean

  // Optionally selects the first char.
  def headOption: Option[Char]

  // The initial part of the string without its last char.
  def init: String

  // Iterates over the inits of this string.
  def inits: Iterator[String]

  // Selects the last char of this string.
  def last: Char

  // Optionally selects the last char.
  def lastOption: Option[Char]

  // Builds a new collection by applying a function to all chars of this string.
  def map[B](f: Char => B): IndexedSeq[B]

  // Builds a new string by applying a function to all chars of this string.
  def map(f: Char => Char): String

  // Returns new sequence with elements in reversed order.
  def reverse: String
```

Corrected

```
  // Selects an interval of elements.
  def slice(from: Int, until: Int): String
  // Tests if this string starts with the specified prefix.
  def startsWith(prefix: String): Boolean
  // Returns a string that is a substring of this string.
  def substring(beginIndex: Int, endIndex: Int): String
  // Iterates over the tails of this string.
  def tails: Iterator[String]
  // A string containing the first n chars of this string.
  def take(n: Int): String
  // ...
```

```
abstract class Option[+A]:
  // Returns the value if the option is nonempty, otherwise returns the
  // given default value.
  def getOrElse[B >: A](default: => B): B
  // Returns true if the option is nonempty and the predicate holds for its
  // value.
  def exists(p: A => Boolean): Boolean
  // Returns true if the option is empty or the predicate holds for its value.
  def forall(p: A => Boolean): Boolean
  // Returns a new option containing the result of applying f to this option's
  // value if it is nonempty. Otherwise, returns None.
  def flatMap[B](f: A => Option[B]): Option[B]
  // Returns the value if the option is nonempty. Otherwise, throws
  // java.util.NoSuchElementException.
  def get: A
  // Returns true if this option is an instance of Some.
  def isDefined: Boolean
  // Returns true if this option is an instance of None.
  def isEmpty: Boolean
  // Applies the given function if this option is nonempty.
  def map[B](f: A => B): Option[B]
  // ...
```

Corrected

Space for rough work.