Profs. Viktor Kunčak, Martin Odersky, and Clément Pit-Claudel
CS-214 Software Construction
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

▩ ☒ ▢ ⊡ ▨ ▢

Deduplication (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.*

Write a function `distinctLast[T](xs: List[T]): List[T]` that takes a list of elements and returns a new list containing all the elements of the original list, but without duplicates. The function should preserve the order of elements: an element `a` can occur before `b` in its output only if this was also the case in its input.

If there are duplicate elements in the original list, only the *last* occurrence of each duplicate should be kept in the output list.

You can not use the `distinct` method from the Scala standard library.

Here are example tests that your implementation must pass:

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

test("distinctLast: list without duplicates"):
  assertEquals(distinctLast(List(1, 2)), List(1, 2))

test("distinctLast: list with duplicates"):
  assertEquals(distinctLast(List(1, 2, 1, 3)), List(2, 1, 3))
```

The runtime complexity of your implementation should not be more than quadratic ($\mathcal{O}(n^2)$).

```
// Complexity: O(n^2)
def distinctLast[T](xs: List[T]) =
  xs.foldRight(List.empty[T])((x, acc) ⇒
    if acc.contains(x) then acc
    else x :: acc
  )

// Complexity: O(n^2)
def distinctLastFoldLeft[T](xs: List[T]) =
  xs.reverse.foldLeft(List.empty[T])((acc, x) ⇒
    if acc.contains(x) then acc
    else x :: acc
  )

// Complexity: O(n^2)
def distinctLastRec[T](xs: List[T]): List[T] =
  xs match
    case Nil ⇒ Nil
    case head :: tail ⇒
      val tailDistinct = distinctLastRec(tail)
      if tailDistinct.contains(head) then tailDistinct
      else head :: tailDistinct

// Complexity: O(n^2)
def distinctLastTailRec[T](xs: List[T]): List[T] =
  @tailrec
  def distinctReverseTailRec(ys: List[T], acc: List[T]): List[T] =
    ys match
      case Nil ⇒ acc
      case head :: tail ⇒
        if acc.contains(head) then distinctReverseTailRec(tail, acc)
```

```
      else distinctReverseTailRec(tail, head :: acc)
distinctReverseTailRec(xs.reverse, Nil)
```

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(a: Int, b: Int): Int =
  if b == 0 then 0
  else a + f(b - 1, a)
```

Write the step-by-step evaluation of the expression `f(2, 2)`:

Technically, only the first solution below is “step-by-step evaluation”, because the second one performs arithmetic in a single step. We gave full points to both.

```
    f(2, 2)
=== 2 + f(2 - 1, 2)
=== 2 + f(1, 2)
=== 2 + (1 + f(2 - 1, 1))
=== 2 + (1 + f(1, 1))
=== 2 + (1 + (1 + f(1 - 1, 1)))
=== 2 + (1 + (1 + f(0, 1)))
=== 2 + (1 + (1 + (0 + f(1 - 1, 0))))
=== 2 + (1 + (1 + (0 + f(0, 0))))
=== 2 + (1 + (1 + (0 + 0)))
=== 2 + (1 + (1 + 0))
=== 2 + (1 + 1)
=== 2 + 2
=== 4
```

```
    f(2, 2)
=== 2 + f(1, 2)
=== 2 + (1 + f(1, 1))
=== 2 + (1 + (1 + f(0, 1)))
=== 2 + (1 + (1 + (0 + f(0, 0))))
=== 2 + (1 + (1 + (0 + 0)))
=== 4
```

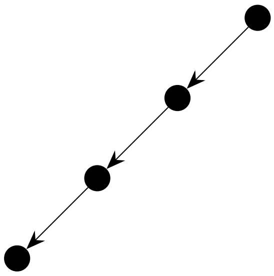
What does `f` do, in one word?

`Multiply, multiplication, multiply a and b, times, product, etc.`

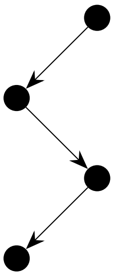
Line trees (14 pts)

We define a *line tree* as a binary tree where each node has either 0 or 1 children, but never 2.

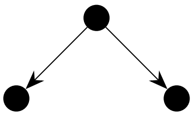
Here are two line trees (1) and (2) and two non-line trees (3) and (4):



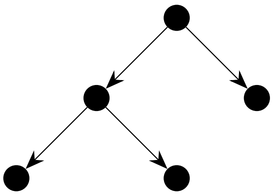
(1)



(2)



(3)



(4)

In this exercise, we use the following case class to represent binary trees:

```
case class MyTree(left: Option[MyTree], right: Option[MyTree])
```

For example, the tree (4) above can be represented as:

```
val tree4 = MyTree(
  Some(MyTree(
    Some(MyTree(None, None)),
    Some(MyTree(None, None))
  )),
  Some(MyTree(None, None))
)
```

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

```
def isLine1(tree: MyTree): Boolean =
  if tree.left.isEmpty then
    if tree.right.isEmpty then true
    else isLine1(tree.right.get)
  else if tree.right.isEmpty then isLine1(tree.left.get)
  else false
```

Question 3 Is `isLine1` correct?

■ Yes ☐ No

```
def isLine2(tree: MyTree): Boolean =
  (tree.left.isEmpty && tree.right.isEmpty)
    || (tree.left.isEmpty && isLine2(tree.right.get))
    || (tree.right.isEmpty && isLine2(tree.left.get))
```

Question 4 Is `isLine2` correct?

■ Yes ☐ No

```
def isLine3(tree: MyTree): Boolean =
  (tree.left.isEmpty && isLine3(tree.right.get))
    || (tree.right.isEmpty && isLine3(tree.left.get))
    || (tree.left.isEmpty && tree.right.isEmpty)
```

Question 5 Is `isLine3` correct?

☐ Yes ■ No

```
def isLine4(tree: MyTree): Boolean =
  if tree.left.isEmpty then (tree.right.isEmpty || isLine4(tree.right.get))
  else if tree.right.isEmpty then isLine4(tree.left.get)
  else false
```

Question 6 Is `isLine4` correct?

■ Yes ☐ No

```
def isLine5(tree: MyTree): Boolean =
  (tree.left.isEmpty && (tree.right.isEmpty || isLine5(tree.right.get)))
    || (tree.right.isEmpty && (tree.left.isEmpty || isLine5(tree.left.get)))
```

Question 7 Is `isLine5` correct?

■ Yes ☐ No

```
def isLine6(tree: MyTree): Boolean =
  tree match
    case MyTree(None, None)     ⇒ true
    case MyTree(Some(left), _)  ⇒ isLine6(left)
    case MyTree(_, Some(right)) ⇒ isLine6(right)
```

Question 8 Is `isLine6` correct?

☐ Yes ■ No

```
def isLine7(tree: MyTree): Boolean =
  tree match
    case MyTree(Some(left), Some(right)) ⇒ false
    case MyTree(Some(left), _)           ⇒ isLine7(left)
    case MyTree(_, Some(right))          ⇒ isLine7(right)
    case _                               ⇒ true
```

Question 9 Is `isLine7` correct?

■ Yes ☐ No

Solutions

Proof of MapSingleFlatten (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`, `xs: List[T]`, `ys: List[T]`, `l: List[T]`, `xss: List[List[T]]` and `f: T => List[T]`.

Given the following lemmas:

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

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

(FlattenNil) `Nil.flatten === Nil`

(FlattenCons) `(xs::xss).flatten === xs ++ xss.flatten`

(ConcatNil) `Nil ++ xs === xs`

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

(Single) `single(x) === x::Nil`

You need to prove:

(MapSingleFlatten) `l.map(single).flatten === l`

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.map(single).flatten === Nil`

```
    Nil.map(single).flatten
=== Nil.flatten              by MapNil
=== Nil                      by FlattenNil
```

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

`(x::xs).map(single).flatten === x::xs`

given that the induction hypothesis, named IH, holds:

(IH) `xs.map(single).flatten === xs`

```
    (x::xs).map(single).flatten
=== (single(x)::xs.map(single)).flatten     by MapCons
=== ((x::Nil)::xs.map(single)).flatten      by Single
=== (x::Nil) ++ xs.map(single).flatten      by FlattenCons
=== (x::Nil) ++ xs                          by IH
=== x::(Nil ++ xs)                          by ConcatCons
=== x::xs                                   by ConcatNil
```

`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.*

A *Pythagorean triple* consists of three positive integers (a, b, c) where $a^2 + b^2 = c^2$. For example, $(3, 4, 5)$ is a Pythagorean triple because $3^2 + 4^2 = 9 + 16 = 25 = 5^2$.

Implement the following function that takes an integer $n < 10^4$ as a parameter and that produces a list of all Pythagorean triples (a, b, c) such that $0 < a \leq b \leq c \leq n$, in at most $\mathcal{O}(n^3)$ time.

The order of the triples in the list does not matter. In other words, if (a_1, b_1, c_1) and (a_2, b_2, c_2) are both valid triples, then it does not matter which one appears earlier in the list.

You must use a `for` comprehension in order to get any points for this question. Your solution must take at most $\mathcal{O}(n^3)$ time in order to get any points for this question.

You do not have to consider the possibility of integer overflow.

```
def pythagoreanTriples(n: Int): List[(Int, Int, Int)] =
```

Given below are three equally valid solutions.

```
(
  for
    a ← 1 to n
    b ← a to n
    c ← b to n
    if a * a + b * b == c * c
  yield
    (a, b, c)
// We need '.toList' to actually return a List!
// (We didn't take points off if you forgot this.)
).toList
```

```
for
  b ← (1 to n).toList // Alternative place for '.toList'.
  a ← 1 to b
  c = Math.sqrt(a * a + b * b)
  // Check that a^2 + b^2 is a perfect square ≤ n
  if c.toInt == c && c ≤ n
yield
  (a, b, c.toInt)
```

```
(
  for
    // 'until' (an exclusive range) is fine for the upper bound of a or b,
    // because there cannot be a Pythagorean triple 0 < a, b, c ≤ n where
    // a or b are actually equal to n.
    a ← 1 until n
    // But for the upper bound of c, we must use 'to', to include c = n.
    c ← (a + 1) to n
    b = Math.sqrt(c * c - a * a)
    if b.isWhole // '.isWhole' is another way to check whether a number is an
  integer.
      // And a must be less than b. We already know 0 < a and b < c ≤ n.
      && b > a
  yield
    (a, b.toInt, c)
).toList
```

Subtyping (14 pts)

Consider the following typing relationships for `Cat`, `Animal`, `Organism` and `Dog`:

- `Cat <: Animal`
- `Animal <: Organism`
- `Dog <: Animal`

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

Question 12 Is it the case that `Dog <: Organism` ?

☒ Yes ☐ No

Question 13 Is it the case that `(Cat ⇒ Organism) <: (Animal ⇒ Dog)` ?

☐ Yes ☒ No

Question 14 Is it the case that `((Organism ⇒ Dog) ⇒ Dog) <: ((Dog ⇒ Organism) ⇒ Cat)` ?

☐ Yes ☒ No

Question 15 Is it the case that
`(Animal ⇒ ((Animal ⇒ Dog) ⇒ Cat)) <: (Cat ⇒ ((Organism ⇒ Dog) ⇒ Organism))` ?

☒ Yes ☐ No

Consider also the following classes:

- **`class`** `List[+T]`
- **`class`** `Sink[-T]`
- **`class`** `Array[T]`

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

Question 16 Is it the case that `List[Organism] <: List[Dog]` ?

☐ Yes ☒ No

Question 17 Is it the case that `Sink[Sink[Organism] ⇒ Organism] <: Sink[Sink[Dog] ⇒ Dog]` ?

☒ Yes ☐ No

Question 18 Is it the case that
`Sink[List[Array[Organism] ⇒ Organism]] <: Sink[List[Array[Dog] ⇒ Dog]]` ?

☐ Yes ☒ No

Fold and 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.

Fold and Permutations

The sequential operation `foldRight` on a `List` processes elements in a fixed order from right to left starting with a known element. However, sometimes, we expect our list elements to arrive in parallel, so we may know nothing about their order! Is it possible that our folding operation produces the same result regardless of the order?

As a concrete representation, consider the operation `isSameFold`:

```
val l1: List[Int]
val l2: List[Int]
val f: (Int, String) ⇒ String
val z: String

val isPermutation = l1.sorted == l2.sorted
val isSameFold = l1.foldRight(z)(f) == l2.foldRight(z)(f)
```

where `l1` and `l2` are permutations of each other, i.e., `isPermutation` is true.

You can find the signature of `List.foldRight` in the appendix for your reference.

Question 19 Which one of the following conditions on the operation is well-formed and sufficient to say that `isSameFold` necessarily holds, assuming that `isPermutation` holds?

☐ $\forall a.\ f(a, z) = a$
☐ $\forall a\ b\ c.\ f(f(a, b), c) = f(f(a, c), b)$
☐ $\forall a\ b\ c.\ f(a, f(b, c)) = f(f(a, b), c)$
☒ $\forall a\ b\ c.\ f(a, f(b, c)) = f(b, f(a, c))$
☐ None, always holds
☐ $\forall a\ b.\ f(a, b) = f(b, a)$

Prime Time

We say a natural number n is prime if and only if its only divisors are 1 and itself. Consider the task of listing all prime numbers less than or equal to a given natural number N. One way to do this is called the *Sieve of Eratosthenes*, which dates back to ancient Greece. The algorithm begins by listing all numbers from 2 up to N. Then, choosing the first number, 2, as a *pivot*, cross out every multiple of it other than itself, as those are divisible by 2, and thus not prime. As a running example, take $N = 17$:

2	3	4	5
6	7	8	9
10	11	12	13
14	15	16	17

$\xrightarrow{\text{pivot} = 2}$

2	3	~~4~~	5
~~6~~	7	~~8~~	9
~~10~~	11	~~12~~	13
~~14~~	15	~~16~~	17

Then, with this filtered grid, choose the next remaining uncrossed number after 2, which is 3, and cross out its multiples as well. One can proceed to do this for all remaining numbers recursively, but it suffices to stop at $\sqrt{N}$, as any composite number up to N has a factor not greater than $\sqrt{N}$.

2	**3**	~~4~~	5
~~6~~	7	~~8~~	9
~~10~~	11	~~12~~	13
~~14~~	15	~~16~~	17

pivot = 3 →

2	**3**	~~4~~	5
~~6~~	7	~~8~~	~~9~~
~~10~~	11	~~12~~	13
~~14~~	~~15~~	~~16~~	17

The remaining numbers are all the prime numbers up to N $(= 17)$:

2	3	5	7	11	13	17

Based on this description of the Sieve of Eratosthenes, we can write a few possible implementations to compute primes up to a given limit. Opportunities for parallelization are plenty! But do they lead to correct behaviour?

An implementation is considered *correct* if and only if for every integer input `upto`, such that $2 \leq$ `upto` $\leq 10^6$, it produces a list of exactly the prime numbers between 2 and `upto`, inclusive on both ends. The order of elements in the output does not matter.

Question 20

Is the implementation `primes1` correct?

```
def primes1(upto: Int): List[Int] =
  val base = (2 to upto).toList
  val limit = math.floor(math.sqrt(upto)).toInt
  val primes =
    (2 to limit).foldLeft(base)((agg, num) ⇒
      agg.filter(p ⇒ p ≤ num || p % num != 0)
    )
  primes
```

■ Yes ☐ No

Question 21

Is the implementation `primes2` correct?

```
def primes2(upto: Int): List[Int] =
  val base = (2 to upto).toList
  val limit = math.floor(math.sqrt(upto)).toInt
  val primes =
    (2 to limit).foldLeft(base.par)((agg, num) ⇒
      agg.filter(p ⇒ p ≤ num || p % num != 0)
    )
  primes.toList // convert parallel collection back to List
```

■ Yes ☐ No

Question 22

Is the implementation `primes3` correct?

```
def primes3(upto: Int): List[Int] =
  val base = (2 to upto).toList
  val limit = math.floor(math.sqrt(upto)).toInt
  val primes = (2 to limit).par
    .aggregate(base)(
      (agg, num) ⇒ agg.filter(p ⇒ p ≤ num || p % num != 0),
      (agg1, agg2) ⇒ agg1.intersect(agg2)
    )
  primes
```

■ Yes ☐ No

Question 23

Is the implementation `primes4` correct?

```
def primes4(upto: Int): List[Int] =
  val base = (2 to upto).toList
  val limit = math.floor(math.sqrt(upto)).toInt
  val primes = base.foldLeft(List[Int]())((agg, num) ⇒
    if agg.forall(num % _ != 0) then num :: agg else agg
  )
  primes
```

■ Yes ☐ No

Question 24

Is the implementation `primes5` correct?

```
def primes5(upto: Int): List[Int] =
  val base = (2 to upto).toList
  val limit = math.floor(math.sqrt(upto)).toInt
  val primes = base.par
    .foldLeft(List[Int]())((agg, num) ⇒
      if agg.forall(num % _ != 0) then num :: agg else agg
    )
  primes
```

■ Yes ☐ No

Question 25

Is the implementation `primes6` correct?

```
def primes6(upto: Int): List[Int] =
  val base = (2 to upto).toList
  val limit = math.floor(math.sqrt(upto)).toInt
  val primes = base.par
    .aggregate(List[Int]())(
      (agg, num) ⇒ if agg.forall(num % _ != 0) then num :: agg else agg,
      (agg1, agg2) ⇒ agg1 ++ agg2
    )
  primes
```

☐ Yes ■ No

Prefix to Postfix (15 points)

Question 26 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.

Your task is to complete the function `prefixToPostfix` that converts an expression from prefix notation (also known as *Polish notation*) to postfix notation (also known as *reverse Polish notation*).

The expressions are represented as lists of `Atom`s, where an `Atom` can either be a number (`Num`) or an operator (`Add` or `Sub`):

```
enum Atom:
  case Num(value: Int)
  case Add
  case Sub
import Atom.*
```

`prefixToPostfix` should return a pair consisting of two lists of `Atom`s:

- The first list is the actual result of the conversion; an expression in postfix form.
- The second list contains any remaining unprocessed elements from `input`

Here are example tests that your implementation must pass successfully:

```
  test("+ 1 2 becomes 1 2 +"):
    val res = prefixToPostfix(List(Add, Num(1), Num(2)))
    assertEquals(res, (List(Num(1), Num(2), Add), Nil))
  test("+ - 3 2 1 becomes 3 2 - 1 +"):
    val res = prefixToPostfix(List(Add, Sub, Num(3), Num(2), Num(1)))
    assertEquals(res, (List(Num(3), Num(2), Sub, Num(1), Add), Nil))
  test("incomplete expression: 1 0 returns 1, and 0"):
    val res = prefixToPostfix(List(Num(1), Num(0)))
    assertEquals(res, (Num(1) :: Nil, Num(0) :: Nil))
```

You should only write in the boxes below, and you cannot declare new `val`s or `def`s.

```
def prefixToPostfix(input: List[Atom]): (List[Atom], List[Atom]) =
  input match
    case Nil              ⇒ (Nil, Nil)
    case Num(value) :: xs ⇒ (Num(value) :: Nil, xs)
    case op :: xs ⇒
      val (output1, xs1) = prefixToPostfix(xs)
      val (output2, xs2) = prefixToPostfix(xs1)
      (output1 ++ output2 ++ List(op), xs2)
```

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
```