EPFL

Profs. Viktor Kunčak, Martin Odersky, and Clément Pit-Claudel
CS-214 Software Construction Midterm
Nov 6, 2024, 16:15–18:15
Duration: 120 minutes

1

Annie Easley

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

Wait for the start of the exam before turning to the next page. This document is printed double-sided, 20 pages. Do not unstaple or detach any pages.

Material	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: 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 pages of this exam contain 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 51 points. The points for each question are indicated next to it.
Stay functional	Unless explicitly stated otherwise, do not use `var`s, `while` loops, `for...do` loops, etc. This will result in 0 points for that question.

1 MSort (10 pts)

Consider the following implementation of an algorithm `mSort`:

```
def mSort(ls: List[Int]): List[Int] =
  ls match
    case Nil => Nil
    case a :: Nil => a :: Nil
    case a :: b :: tail =>
      val (ll, lr) = split(tail)
      merge(mSort(ll), mSort(lr))
```

The functions `split`, `merge`, and `isSortedAscending` are provided as the summaries below:

```
/** Check if a given list is sorted */
def isSortedAscending(ls: List[Int]): Boolean =
  ls == ls.sortWith(_ < _)

/** Given two sorted lists, merge them into a new sorted list */
def merge(ll: List[Int], lr: List[Int]): List[Int] = {
  require(isSortedAscending(ll) && isSortedAscending(lr))
  // ...
}.ensuring { l =>
  l == (ll ++ lr).sortWith(_ < _)
}

/** Split a list into two lists containing the first and second halves respectively */
def split[A](ls: List[A]): (List[A], List[A]) = {
  // ...
}.ensuring { case (ll, lr) =>
  ls == ll ++ lr && ll.length == ls.length / 2
}
```

Question 1 This part is worth 1 point.

☐0 ☐1 *Do not write here.*

Compute the output of `mSort` on the list `List(5, 2, 3, 1, 4)`.

Given the implementation of `mSort` as above, for each of the following properties, indicate whether it is true for all lists `ls` of type `List[Int]`:

Question 2 `mSort(ls).length <= 1` *(1 point)*

☐ Yes ☐ No

Question 3 `mSort(ls).length <= 2` *(1 point)*

☐ Yes ☐ No

Question 4 `isSortedAscending(mSort(ls))` *(1 point)*

☐ Yes ☐ No

Question 5 `mSort(ls).length == ls.length` *(1 point)*

☐ Yes ☐ No

Question 6 `mSort(ls).length <= ls.length` *(1 point)*

☐ Yes ☐ No

Question 7 This part is worth 4 points.

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

`mSort` is supposed to sort its input list, but the given implementation is faulty. Change **one line** to make it correctly sort the list: choose one line, cross it out, and write its replacement below it. In your new line of code, do not introduce calls to functions that are not already called in some way in the body of `mSort`.

```
def mSort(ls: List[Int]): List[Int] =

  ls match

    case Nil => Nil

    case a :: Nil => a :: Nil

    case a :: b :: tail =>

      val (ll, lr) = split(tail)

      merge(mSort(ll), mSort(lr))
```

2 Proof (8 pts)

All lemmas on this page hold for all types `T` and `S` and all `z` of type `S`, `f` of type `(S, T) => S`, `x` of type `T`, `xs` of type `List[T]`, `ls1` of type `List[T]`, and `ls2` of type `List[T]`.

Given the following lemmas:

(FoldLeftNil)	`Nil.foldLeft(z)(f) === z`
(FoldLeftCons)	`(x :: xs).foldLeft(z)(f) === xs.foldLeft(f(z, x))(f)`
(NilAppend)	`Nil ++ xs === xs`
(AppendNil)	`xs ++ Nil === xs`
(ConsAppend)	`(x :: xs) ++ ys === x :: (xs ++ ys)`
(AppendCons)	`xs ++ (y :: ys) === (xs ++ (y :: Nil)) ++ ys`
(ReverseNil)	`Nil.reverse === Nil`
(ReverseCons)	`(x :: xs).reverse === xs.reverse ++ (x :: Nil)`
(FlipConsEval)	`flipCons(xs, x) === x :: xs`

Your task is to prove FoldRevAppend:

```
ls1.foldLeft(ls2)(flipCons) === ls1.reverse ++ ls2
```

Complete the proof below. You must write every step of reasoning. For each step, you may only use one lemma, and you must write the name of the lemma you are using. You may only use the lemmas above, or the induction hypothesis in the inductive case. The proof is done by induction on `ls1`.

Question 8 This part is worth 3 points.

☐0 ☐1 ☐2 ☐3 *Do not write here.*

Base case: `ls1` is `Nil`. Therefore, you need to prove:

```
Nil.foldLeft(ls2)(flipCons) === Nil.reverse ++ ls2
```

Question 9 This part is worth 5 points.

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

Inductive case: `ls1` is `x :: xs`. Therefore, you need to prove:

```
(x :: xs).foldLeft(ls2)(flipCons) === (x :: xs).reverse ++ ls2
```

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

```
xs.foldLeft(y)(flipCons) === xs.reverse ++ y
```

3 Shiritori (11 pts)

In the game called shiritori, players collaboratively take turns to construct a sequence of nonempty words in which each word begins with a nonempty suffix of the previous one. The game finishes when a player says the word "`end`".

Example

The following is a *complete*, *nonrepeating*, *valid* shiritori sentence:
`scala latch check checkbook book bookshelf flame amend end`

Definitions

We say a sentence is *complete* whenever its last word is `end`. Hence, the example above is complete.
We say that a sentence is *nonrepeating* whenever it does not use the same word twice. Hence, the example above is nonrepeating.
We say that a sentence is *valid* whenever it respects the rule that each word must begin with a non-empty suffix of the previous one. Hence, the example above is valid, as shown in the diagram below:

```
scala
   latch
      check
      checkbook
           book
           bookshelf
                   flame
                     amend
                       end
```

Nonexamples

In contrast to the example above, the following sentence is complete, but *invalid*, because `drop` may not follow `network` in a valid shiritori sentence:
`internet network drop rod depend end`

The following sentence is valid, but *incomplete*, because its last word is not `end`:
`polymorphism morphism small alley eye yearn`

And the following sentence is valid and complete, but *repeating*, because it uses the word `koala` twice:
`scala latch check koala asterisk koala attend end`

Objective

The objective of this exercise is to write a function to construct all complete, nonrepeating, and valid shiritori sentences, using words from a given set of words and starting with a given word.
We call such sentences *winning sentences.*

We define the following types to reason about shiritori sentences:

```scala
type Word = String
type Sentence = List[Word]
```

In the following, you may assume that all words given as inputs are exclusively composed of lowercase English letters ('a' to 'z').

3.1 Checking shiritori sentences

Question 10 This part is worth 3 points.

☐0 ☐1 ☐2 ☐3 *Do not write here.*

Write a function `isValidStep` such that `isValidStep(first, second)` is true if and only if the word `second` may follow the word `first` in a shiritori sentence. In particular, your function should pass the following tests:

```
test("isValidStep"):
  assertEquals(isValidStep("koala",     "asterisk"),  true)
  assertEquals(isValidStep("scala",     "latch"),     true)
  assertEquals(isValidStep("checkbook", "book"),      true)
  assertEquals(isValidStep("book",      "book"),      true)
  assertEquals(isValidStep("book",      "bookshelf"), true)
  assertEquals(isValidStep("scala",     "java"),      false)
  assertEquals(isValidStep("drop",      "network"),   false)
```

```
def isValidStep(first: Word, second: Word): Boolean =
  require(first.nonEmpty && second.nonEmpty)
```

Question 11 This part is worth 3 points.

0 1 2 3 *Do not write here.*

Write a function `isValidSentence` that checks whether a given shiritori sentence is valid. In particular, in addition to the examples given in the introduction, your function should pass the following tests:

```
test("isValidSentence"):
  assertEquals(isValidSentence(List()),                                  true)
  assertEquals(isValidSentence(List("scala")),                           true)
  assertEquals(isValidSentence(List("scala","java")),                    false)
  assertEquals(isValidSentence(List("acknowledge","edge","geneva")), true)
```

```
def isValidSentence(s: Sentence): Boolean =
  require(s.forall(_.nonEmpty))
```

3.2 Making winning shiritori sentences

Question 12 This part is worth 5 points.

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

Complete the function `allWinningSentences` below, which, given a starting word `first` and a set of words `wordset`, returns the set of *all* winning sentences starting with `first` and composed of words found in `wordset`. In particular, your function should pass the following tests:

```
test("allWinningSentences"):
  assertEquals(
    allWinningSentences("banana", Set("banana","end","anagram")),
    Set())
  assertEquals(
    allWinningSentences("end", Set("end","endgame")),
    Set(List("end")))
  assertEquals(
    allWinningSentences("banana", Set("banana","end","amend")),
    Set(List("banana","amend","end")))
  assertEquals(
    allWinningSentences("rebar", Set("rebar","bartend","send","barstool","toolkits","end")),
    Set(List("rebar","bartend","end"),
        List("rebar","barstool","toolkits","send","end")))
```

```
def allWinningSentences(first: Word, wordset: Set[Word]): Set[Sentence] =
  require(first.nonEmpty)
  require(wordset.contains(first))
  if first == "end" then
    Set(List(first))
  else
    for
```

```
    yield
```

4 Variance (6 pts)

Given below are two faulty class definitions, `C[+T]` and `D[-T]`, that are rejected by the Scala compiler due to variance checks. For each, construct a program which would fail at runtime *if* the Scala compiler were to accept that class definition without variance checks.

You may only use the following in your answer:

- the classes `A` and `B` defined below and their methods;
- the classes and methods defined in each question;
- usual constructions and applications of Scala functions and values; and
- the types `Int`, `String`, `Boolean`, and `Any`, and their methods.

```
class A:
  def foo: A = this

class B extends A:
  def bar: B = this
```

Question 13 This part is worth 3 points.

☐0 ☐1 ☐2 ☐3 *Do not write here.*

```
case class C[+T](f: T => T)
```

is rejected by the compiler with the following message:

```
[error] 7 |  case class C[+T](f: T => T)
[error]   |                   ^^^^^^^^^
[error]   |covariant type T occurs in contravariant position in type T => T of value f
```

Question 14 This part is worth 3 points.

0 1 2 3 *Do not write here.*

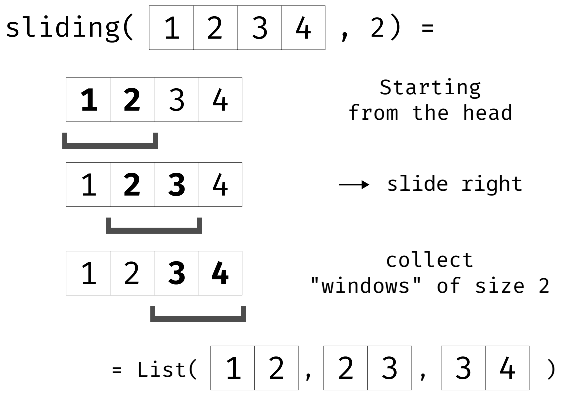
```
class D[-T](a: T):
  def map[U](f: T => U): D[U] = D(f(a))
```

is rejected by the compiler with the following message:

```
[error] 18 |    def map[U](f: T => U): D[U] = D(f(a))
[error]    |                 ^^^^^^^^^
[error]    |contravariant type T occurs in covariant position in type T => U of parameter f
```

5 Sliding Windows (8 pts)

Consider a function `sliding[T]: (ls: List[T], n: Int) => List[List[T]]`, which, given a list `ls` and an integer `n` greater than zero, returns "sliding windows" over `ls`, i.e., every contiguous sublist of `ls` of size `n`, in order, starting from the head.



The output of `sliding[T](ls, n)` must satisfy the following specification:

```scala
def slidingSpec[T](ls: List[T], n: Int): List[List[T]] = {
  require(n > 0)
  // ...
}.ensuring { (windows: List[List[T]]) =>
  (windows.length == math.max(ls.length - n + 1, 0)) &&
  (0 until windows.length).forall(i => windows(i) == ls.slice(i, i + n))
}
```

Note that a correct implementation of `sliding` must pass at least the following tests. They are not complete.

```scala
test("T0: Window smaller than list (worked out) (ls.length = 4, window size = 2)"):
  assertEquals(
    sliding(List(1, 2, 3, 4), 2),
    List(List(1, 2), List(2, 3), List(3, 4))
  )
test("T1: Window smaller than list (ls.length = 4, window size = 3)"):
  assertEquals(
    sliding(List(true, false, true, true), 3),
    List(List(true, false, true), List(false, true, true))
  )
test("T2: Empty list (ls.length = 0, window size = 3)"):
  assertEquals(sliding(List(), 3), List())
test("T3: Window larger than list (ls.length = 3, window size = 4)"):
  assertEquals(sliding(List(1, 2, 3), 4), List())
test("T4: Window same size as list (ls.length = 4, window size = 4)"):
  assertEquals(sliding(List(1, 2, 3, 4), 4), List(List(1, 2, 3, 4)))
```

Question 15 *(4 points)*

The following implementation of `sliding`, `sliding1`, is possibly faulty.

```
def sliding1[T](ls: List[T], n: Int): List[List[T]] =
  require(n > 0)
  if ls.length < n then Nil
  else ls.take(n) :: sliding1(ls.drop(n), n)
```

Check all options that apply to `sliding1`.

- ☐ This implementation is correct.
- ☐ This implementation is incorrect, and test `T0` fails.
- ☐ This implementation is incorrect, and test `T1` fails.
- ☐ This implementation is incorrect, and test `T2` fails.
- ☐ This implementation is incorrect, and test `T3` fails.
- ☐ This implementation is incorrect, and test `T4` fails.
- ☐ This implementation is incorrect, but all provided tests pass.

For each of the following implementations, answer whether it correctly implements `sliding` as specified above.

Question 16 *(1 point)*

```
def sliding2[T](ls: List[T], n: Int): List[List[T]] =
  require(n > 0)
  if ls.length < n then Nil
  else ls.take(n) :: sliding2(ls.tail, n)
```

☐ Yes ☐ No

Question 17 *(1 point)*

```
def sliding3[T](ls: List[T], n: Int): List[List[T]] =
  require(n > 0)
  ls
    .drop(n)
    .foldLeft(List(ls.take(n))): (acc, next) =>
      (acc.head.tail :+ next) :: acc
    .reverse
```

☐ Yes ☐ No

Question 18 *(1 point)*

```
def sliding4[T](ls: List[T], n: Int): List[List[T]] =
  require(n > 0)

  ls
    .foldRight(List(Nil): List[List[T]]): (next, acc) =>
      val nextHead = next :: acc.head
      if nextHead.length == n then nextHead.init :: nextHead :: acc.tail
      else nextHead :: acc.tail
    .tail
```

☐ Yes ☐ No

Question 19 *(1 point)*

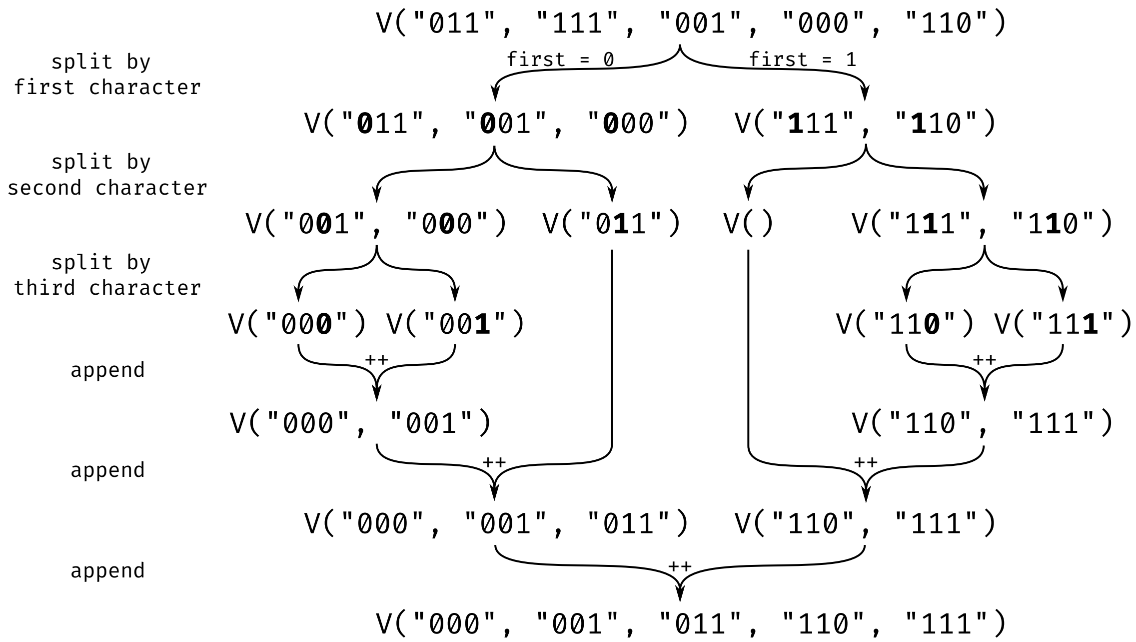
```
def sliding5[T](ls: List[T], n: Int): List[List[T]] =
  (0 until math.max(ls.length - n + 1, 0))
    .map(idx => ls.slice(idx, idx + n))
    .toList
```

☐ Yes ☐ No

6 Parallel Sorting (8 pts)

Radix sort is an algorithm to sort data in increasing alphabetical order (also called “lexicographic order”) without using any explicit comparisons. We restrict ourselves to the case of bitstrings, i.e. strings containing only `0`s and `1`s, with the ordering `0` < `1`. So, `"00"` < `"01"` < `"10"` < `"11"`. We assume that all bitstrings to be sorted are of the same fixed length w.

Radix sort works by splitting the vector of input bitstrings by digits. First, the vector is partitioned into bitstrings starting with `0` and bitstrings starting with `1`. As our ordering is lexicographic, we know that all bitstrings starting with `0` are smaller than those starting with `1`. Then, we recursively sort each partition. Finally, we concatenate (append) the results. The process is illustrated in the diagram below (where `V` stands for `Vector`).



Since the sorting for bitstrings starting with `0` and those starting with `1` are independent, we can parallelize them. Below we show an implementation of a parallel radix sort for bitstrings.

```
def radixSort(vs: Vector[String], startIndex: Int = 0): Vector[String] =
  require(vs.forall(s => s.length == w && s.forall(c => c == '0' || c == '1')))
  require(0 <= startIndex && startIndex <= w)

  if startIndex >= w || vs.length <= 1 then vs
  else
    val (zeroes, ones) = splitByDigit(vs, startIndex)
    val (sortedZeroes, sortedOnes) =
      parallel(
        radixSort(zeroes, startIndex + 1),
        radixSort(ones, startIndex + 1)
      )
    sortedZeroes ++ sortedOnes

def splitByDigit(vs: Vector[String], index: Int): (Vector[String], Vector[String]) =
  (
    vs.filter(s => s(index) == '0'),
    vs.filter(s => s(index) == '1')
  )
```

Given the above implementation of `radixSort` over bitstrings, answer the following questions. Assume that:

- `splitByDigit(vs, at)` runs in time $\mathcal{O}($`vs.length`$)$,
- `vs1 ++ vs2` runs in time $\mathcal{O}(1)$ for vectors `vs1`, `vs2`,
- `vs.length` runs in time $\mathcal{O}(1)$ for a vector `vs`, and
- the **`require`** statements are not executed and do not contribute to the complexity.

Note that the bitstrings in the input vector `vs` may repeat. Let $\lfloor\cdot\rfloor$ denote the floor function returning an integer such that $x - 1 < \lfloor x \rfloor \leq x$.

Question 20 *(2 points)*

While sorting a vector of n elements, what is the maximum number of elements in vector `zeroes` within the function `radixSort`?

☐ n ☐ 2^w ☐ $n-1$ ☐ w ☐ $\lfloor\frac{n}{2}\rfloor + 1$

Question 21 *(2 points)*

Which of the following recurrence equations, when solved, provide a correct upper bound on the worst-case work $W(n,w)$ and depth $D(n,w)$, for sorting a vector of n bitstrings of size w? d_1, d_2, c_1, c_2 are constant values. Assume that both the work and the depth equal a constant c_0 whenever $w = 0$ or $n = 0$.

☐
$$W(n,w) = d_1 n + \max_{0 \leq n_0 \leq n} \big(W(n_0, w-1) + W(n - n_0, w-1)\big) + c_1$$
$$D(n,w) = d_2 n + \max_{0 \leq n_0 \leq n} \max\big(D(n_0, w-1), D(n - n_0, w-1)\big) + c_2$$

☐
$$W(n,w) = d_1 n + \max_{0 \leq n_0 \leq n} \big(W(n_0, w-1) + W(n - n_0, w-1)\big) + c_1$$
$$D(n,w) = d_2 n + \max_{0 \leq n_0 \leq n} \min\big(D(n_0, w-1), D(n - n_0, w-1)\big) + c_2$$

☐
$$W(n,w) = d_1 n + W\left(\left\lfloor\frac{n}{2}\right\rfloor + 1, w-1\right) + c_1$$
$$D(n,w) = d_2 n + D\left(\left\lfloor\frac{n}{2}\right\rfloor + 1, w-1\right) + c_2$$

☐
$$W(n,w) = d_1 n + 2W\left(\left\lfloor\frac{n}{2}\right\rfloor + 1, w-1\right) + c_1$$
$$D(n,w) = d_2 n + D\left(\left\lfloor\frac{n}{2}\right\rfloor + 1, w-1\right) + c_2$$

Question 22 *(2 points)*

What is the worst-case depth $D(n,w)$ for sorting a vector of n bitstrings, each of size w?

☐ $\mathcal{O}(nw)$ ☐ $\mathcal{O}(nw\log n)$ ☐ $\mathcal{O}(n^2 w)$ ☐ $\mathcal{O}(n+w)$

Question 23 *(2 points)*

What is the worst-case work $W(n,w)$ for sorting a vector of n bitstrings, each of size w?

☐ $\mathcal{O}(nw\log n)$ ☐ $\mathcal{O}(nw)$ ☐ $\mathcal{O}(n^2 w)$ ☐ $\mathcal{O}(n+w)$

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
```

```
  // 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]

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

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

  // 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]

  // 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

  // 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]

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

  // 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]

  // ...
```

```scala
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

  // 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

  // ...
```