+1/1/60+

EPFL

Profs. Viktor Kunčak, Martin Odersky, and Clément Pit-Claudel
CS-214 Mini Mock Midterm (1 exercise only)
Nov 6, 2024
Duration: 25 minutes

1

Annie Easley

SCIPER : **1000001** Room : **At home** Signature :

Wait for the start of the exam before turning to the next page. This document is printed double-sided, 6 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 9 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.

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 Truthy Truths (9 pts)

Consider the following definition of propositional formulas defined as an enum `Formula`:

```
enum Formula:
  case Variable(id: Int)
  case Not(inner: Formula)
  case And(left: Formula, right: Formula)
  case Or(left: Formula, right: Formula)
import Formula.*
```

We would like to be able to evaluate and reason about these expressions. So, we need to assign *truth values* to variables in our formulas. Such a mapping is an *assignment*:

```
type Assignment = Map[Int, Boolean]
```

Question 1 This part is worth 2 points.

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

Given an `assignment` that assigns *all* the variables in an expression `expr`, complete the function `eval` below to fully evaluate `expr`:

```
def eval(expr: Formula, assignment: Assignment): Boolean =
  expr match
    case Variable(id)     =>
    case Not(inner)       =>
    case And(left, right) =>
    case Or(left, right)  =>
```

Question 2 This part is worth 4 points.

☐0 ☐1 ☐2 ☐3 ☐4 *Do not write here.*

Fill in the **for**-comprehension below to complete the function `assignments` to generate *all* possible assignments for a given list of variables:

```
def assignments(variables: List[Variable]): List[Assignment] =
  variables match
    case Nil => List(Map.empty)
    case variable :: rest =>
      val partialAssignments = assignments(rest)
      for
        ???
      yield
        ???
```

Question 3 This part is worth 3 points.

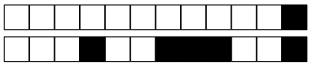
☐0 ☐1 ☐2 ☐3 *Do not write here.*

Now, we are equipped to generate *all possible evaluations* for a given expression and list of variables. We call a listing of all such assignments and their evaluations a *truth table* for an expression.

Complete the function `truthTable` to generate a truth table for a given expression. Assume that `variables` contains at least all variables present in `expr`.

You may use the functions defined in the previous parts. Keep your code succinct.

```
def truthTable(expr: Formula, variables: List[Variable]): List[(Assignment, Boolean)] =
  ???
```




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]
  // ...
```

```
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
  // ...
```