MIT
6.102 Software Construction
Prof. Rob Miller

6.102 Spring 2023 Quiz 2

You have **75** minutes to complete this quiz. There are **6** problems, each worth approximately an equal share of points.

The quiz is **closed-book** and closed-notes, but you are allowed one two-sided page of notes on paper, and you may use blank scratch paper. You may not open or use anything else on your computer: no 6.102 website or readings; no VS Code, TypeScript compiler, or programming tools; no web search or discussion with other people.

Before you begin: you must *check in* by having the course staff scan the QR code at the top of the page.

To leave the quiz early: click the *done* button at the very bottom of the page and show your screen with the check-out code to a staff member.

This page automatically saves your answers as you work. If you see a stuck yellow spinner, red exclamation mark, or a red notification that you are disconnected, your answers are not being saved: try reloading the page right away, before continuing to work on the quiz.

If you feel the need to write a note to the grader, you can click the gray pencil icon to the right of the answer.

Good luck!

You can **open this introduction in a separate tab**.

The questions in this quiz are about an immutable abstract data type `PointSet`, which represents a set of points in the 2D plane. Examples of point sets include:

- all points inside the circle centered at (0.5, -1) with radius 2.3
- the union of all points inside a group of rectangles
- the x axis (the points (0,y) for all y)
- the origin (0,0)

`PointSet` is shown in the code below.

The `contains` instance method is one operation of `PointSet`.

The `disk`, `intersect`, and `union` functions are *also* operations of `PointSet`, even though they are not instance methods. In the code below, their specs may be incomplete.

`PointSet` may have other operations as well, not shown in the code below.

```
// Represents an immutable set of points in the 2D plane
interface PointSet {

    /** @returns true if and only if point is an element of this set */
    contains(point: Point): boolean;

}

// returns the set of points inside, or on the edge of, a circle [spec may be incomplete]
function disk(center: Point, radius: number): PointSet;

// returns the set of points in the intersection of two point sets [spec may be incomplete]
function intersect(set1: PointSet, set2: PointSet): PointSet;

// returns the set of points in the union of two point sets [spec may be incomplete]
function union(set1: PointSet, set2: PointSet): PointSet;



// Represents an immutable point in the 2D plane
class Point {
    constructor(
        public readonly x: number,
        public readonly y: number
    ) {
        /* no code here */
    }
}
```

Problem ⤫1.

What kind of `PointSet` ADT operation is `disk`?

creator

What kind of `PointSet` ADT operation is `intersect`?

producer

Complete the specification for `disk`. Your answer should fit in the box without scrolling.

```
/**
```

```
@param center of circle
@param radius of circle, should be nonnegative
@returns the set of points inside or on edge of the circle centered at `center` with radius
`radius`
```

```
*/
function disk(center: Point, radius: number): PointSet;
```

Suppose you use variant classes to implement the `disk`, `intersect`, and `union` operations.

Write a data type definition for `PointSet` with only these three variant classes. Your answer should fit in the box without scrolling.

```
PointSet = Disk(center: Point, radius: number)
         + Intersect(set1: PointSet, set2: PointSet)
         + Union(set1: PointSet, set2: PointSet)
```

Implement `intersect` using a variant class of `PointSet` by filling in the boxes below. Write just TypeScript code **with no comments or documentation**. Your answers should fit in the boxes without scrolling.

`Intersect` variant class:

```
class Intersect implements PointSet {
  constructor(private readonly set1: PointSet, private readonly set2: PointSet) { }
  public contains(point: Point): boolean {
    return this.set1.contains(point) && this.set2.contains(point);
  }
}
```

`intersect()` function:

```
function intersect(set1: PointSet, set2: PointSet): PointSet {
```

```
return new Intersect(set1, set2);
```

```
}
```

Problem ⤫2.

For the `Union` variant, Benevolent Bitdiddle notices that the program speeds up a lot if he changes its `contains` operation as follows:

```
// original Union contains() operation
public contains(point: Point): boolean {
    return this.set1.contains(point) || this.set2.contains(point);
}

// Ben's new version
public contains(point: Point): boolean {
1    if ( this.set1.contains(point) ) return true;
2    if ( this.set2.contains(point) ) {
3       // swap set1 and set2
4       const oldset1 = this.set1;
5       this.set1 = this.set2;
6       this.set2 = oldset1;
7       return true;
8    }
9    return false;
}
```

Ben says: “This reorganizes a tree of `Union` nodes so that sets that are more likely to contain points will be checked first, while sets that are less likely may not need to be checked at all.”

Write the abstraction function for `Union`, and use it to argue that Ben’s new version is a benevolent side effect. Your answer should fit in the box without scrolling.

AF(set1, set2) = the set of points that are the union of set1 and set2.
Union is commutative, so AF(a,b) = AF(b,a), so the order of set1 and set2 doesn't matter to the abstract value, which is why Ben's side effect is benevolent.

Louis Reasoner writes a code-review comment on line 5 of Ben’s code, complaining that it breaks the rep invariant temporarily. Respond to Louis’s comment by writing the rep invariant for `Union` and explaining why he is right, or wrong, or partly-right (e.g. if line 5 does break *something* temporarily). Your answer should fit in the box without scrolling.

RI(set1, set2) = true. (The rep needs no additional constraints beyond the static types of set1 and set2.)
Louis is wrong to say that line 5 doesn't break the rep invariant, but he is right that line 5 temporarily changes the abstract value that the rep corresponds to, which temporarily breaks the immutability of the type.

Alyssa Hacker chimes in with a code-review comment saying that the safety-from-rep-exposure argument needs to be updated. For the original version of the code, the SRE argument was this:

```
// all rep fields are private, immutable, and unreassignable
```

What should it be for Ben’s version?

all rep fields are private and immutable

Louis joins the rep-exposure thread, and suggests reviewing the ways that a client might conceivably get an alias to an object in the rep. List all the types that are used as parameters or return values of `Union` operations:

Point, boolean, PointSet

Which of these types cannot be involved in aliasing between a client and the rep of `Union`? Why? Your answer should fit in the box without scrolling, and write "none" if none of the types apply.

Point and boolean are not found in the rep of Union, so can't be involved in aliasing between client and rep. (Union's rep consists of set1 and set2, which have type PointSet, not Point or boolean.)

Which of these types *might* be involved in aliasing between a client and the rep of `Union`? Explain whether or not they are safe from rep exposure. Your answer should fit in the box without scrolling, and write "none" if none of the types apply.

PointSet could be aliased between client and rep, but it is safe from rep exposure because it is immutable.

Louis Reasoner really likes Ben's hack, and he decides to try it in Python. He notices that Python 3.11 has a `pointset` library with an API very similar to what the team is writing in TypeScript. Here is the spec for its `contains` operation:

```
# defined in module pointset
def contains(pointset, point):
    """returns true if and only if point is a member of pointset"""
```

By exploring with the Python REPL (read-eval-print loop) and printing various `pointset` objects, Louis discovers that in Python 3.11, a union is just a two-element list. So he writes his own version of `contains` that implements Ben's benevolent side effect:

```
from pointset import contains
def my_contains(pointset, point):
    """returns true if and only if point is a member of pointset"""
    if isinstance(pointset, list):
        if my_contains(pointset[0], point):
            return True
        if my_contains(pointset[1], point):
            # swap the elements of the list
            pointset[0], pointset[1] = pointset[1], pointset[0]
            return True
        return False
    else:
        return contains(pointset, point)
```

Louis is thrilled, because his `my_contains` function not only works, but speeds up his program dramatically.

State two different ways that `my_contains` is not ready for change (RFC). Your answers should fit in the boxes without scrolling.

The "isinstance(pointset, list)" is not ready for pointset to change to use lists for something else (e.g. intersection).

Using just pointset[0] and pointset[1] is not ready for lists with more or less than 2 elements.

Name the principle of abstract data type theory that Louis is violating.

representation independence

Problem ⤫3.

For each of the following proposed subtypes of `PointSet`, say whether it is actually a subtype or not, and why. Your answers should fit in the boxes without scrolling.

```
// Represents a mutable set of points in the 2D plane.
interface MutablePointSet extends PointSet {
    /** @inheritdoc */
    contains(point: Point): boolean;

    /** Adds a point to this set. */
    add(point: Point): void;
}
```

Not a subtype, because MutablePointSet is mutable, but PointSet's spec requires immutability.

```
// Represents an immutable set of 2D points with integer coordinates in the 2D plane.
interface IntegerPointSet extends PointSet {
    /**
     * @param point must have integer-valued coordinates
     * @return true if and only if point is in this set
     */
    contains(point: Point): boolean;
}
```

Not a subtype, because the contains() precondition has been strengthened, which means IntegerPointSet has a weaker spec than PointSet.

```
// Represents an immutable set of 2D points, with subsets labeled by labels of an arbitrary type L,
// where L uses === for equality.
interface LabeledPointSet<L> extends PointSet {
    /**
     * @inheritDoc
     */
    contains(point: Point): boolean;

    /**
     * @returns subset of this set that has the label `label`
     */
    get(label: L): PointSet;
}
```

Is a subtype, because LabeledPointSet all the operations (with the same specs) as PointSet, and just strengthens the spec by adding an additional operation.

Problem ⤨4.

Alyssa P. Hacker (being a Lisp Hacker and a functional programmer at heart) proposes an alternative rep for `PointSet`: just storing a *containment predicate* function that determines whether a point is in the set or not. The containment predicate exactly corresponds to the `contains` operation of the set.

For this problem only, the team is considering Alyssa's proposal that there should be *no variant classes* for `PointSet`.

She offers the following new public operation for `PointSet`, which constructs her rep:

```
/**
 * @param f containment predicate; must be a pure function
 *              (f(p) always returns the same value for a given point p)
 * @returns the set of points p for which the containment predicate f(p) is true
 */
function predicate(f: _________): PointSet {
    return { contains: f }; // creates an object whose contains() method is f()
}
```

This code typechecks in TypeScript. The object `{contains:f}` is a legal instance of the `PointSet` interface, because it is an object with a `contains` method that has the right type signature.

But Alyssa observes that `contains` is not really an instance method anymore (it doesn't use `this` internally) but just a simple function. So within implementation code, the containment predicate for any `set` can be retrieved using the expression `set.contains`.

Alyssa also offers some utility functions that she keeps around for operating on functions:

```
const and =     (f, g) => (x) => f(x) && g(x);
const or =      (f, g) => (x) => f(x) || g(x);
const butnot =  (f, g) => (x) => f(x) && !g(x);
const compose = (f, g) => (x) => f(g(x));
```

Fill in the blank in the mathematical type signature for `predicate` below:

`predicate`: ____________ → `PointSet`

(Point -> boolean)

Use `predicate` to implement `intersect`. Your answer should fit in the box without scrolling.

```
function intersect(set1: PointSet, set2: PointSet): PointSet {
  return predicate(and(set1.contains, set2.contains));
}
```

Jazzed by Alyssa's idea, Louis decides to implement `filter` for `PointSet`:

```
/**
 * @param set point set
 * @param f  filter function
 * @return `set` filtered to keep only points that satisfy `f`
 */
```

```
function filter(set:PointSet, f:________): PointSet {
    return predicate( (point: Point) => {
        if (set.contains(point)) {
            if (f(point) === true) {
                return true;
            } else {
                return false;
            }
        } else {
            return false;
        }
    });
}
```

Rewrite Louis's code to make it **as short as you can**. You can use any functions that have been defined, but no variant classes.

```
/**
 * @param set point set
 * @param f  filter function
 * @return `set` filtered to keep only points that satisfy `f`
 */
function filter(set:PointSet, f:________): PointSet {

  return intersect(set, predicate(f));

}
```

Problem ⤫5.

Suppose we add an operation that creates a point set where the actual set is stored on a web server:

```
// make a set of points where the server at `url` decides whether a point is in the set
function remote(url: string): PointSet;
```

We implement `remote` using a new variant class `Remote`. To test whether a point is in the set, the `contains` operation of `Remote` makes a call to the web server, which returns either `'yes'` or `'no'`. Here is the code we try to write for `contains`:

```
public contains(point: Point): boolean {
    const url: string = this.makeUrl(point);
    const response: Response = fetch(url);
    return response.text() === 'yes';
}
```

Unfortunately this code doesn't compile, because both `fetch()` and `response.text()` are asynchronous operations that return promises.

Rewrite this `contains` code to make it compile. Your answer should fit in the box without scrolling.

```
public async contains(point: Point): Promise<boolean> {
  const url: string = this.makeUrl(point);
  const response: Response = await fetch(url);
  return await response.text() === 'yes';
}
```

After making these changes, we now have another problem with `Remote` and `PointSet`. Explain this problem in terms of subtyping. Your answer should fit in the box without scrolling.

After changing contains() to return Promise<boolean> instead of boolean, Remote is no longer a subtype of PointSet, which requires contains() to return a boolean.

Assume that `PointSet` has now been changed to address the subtyping problem. Revisit your answer to the `Intersect` variant in Problem 1, and rewrite just the `contains` method for `Intersect` so that it is **as concurrent as possible**, while still waiting for all promises to resolve. Your answer should fit in the box without scrolling.

```
public async contains(point: Point): Promise<boolean> {
  const promise1 = this.set1.contains(point);
  const promise2 = this.set2.contains(point);
  return await promise1 && await promise2;
}
```

In light of the changes to `PointSet`, Ben Bitdiddle reviews his benevolent-side-effect optimization of `Union`:

```
// Ben's version of Union.contains()
public contains(point: Point): boolean {
    if ( this.set1.contains(point) ) return true;
    if ( this.set2.contains(point) ) {
      // swap set1 and set2
      const oldset1 = this.set1;
      this.set1 = this.set2;
      this.set2 = oldset1;
```

```
        return true;
    }
    return false;
}
```

Assume Ben makes the necessary edits to his code so that it compiles with the new `PointSet`. Ben realizes his code has a concurrency bug.

Name the kind of concurrency bug the code has:

race condition

Describe an interleaving between two concurrent functions A and B, both calling `contains()`, that shows the bug. Your answer should fit in the box without scrolling.

A and B both call contains() on the same Union object. A calls set1.contains() and loses control at the await. B executes completely and swaps set1 and set2. A now resumes, with set1.contains() resolving to false, and proceeds to call set2.contains() -- but because the sets were swapped, it is now checking the same set it originally did, which will produce the wrong answer if the point actually *is* found in the original set2.

Problem ⤮6.

Write a grammar that matches **TypeScript expressions** that use the `union()` and `disk()` operations to make a set consisting of one or more disks, where the disks must be radius 1 with centers at integer coordinates.

Here are two examples of expressions that your grammar should match:

```
disk(new Point(5,5),1)
union( union( disk(new Point(0,0),1) , disk(new Point(2,0),1) ) , disk(new Point(0,2),1) )
```

Your grammar must match all ways to combine using `union`, and all disks with radius 1 and centers at integer coordinates, not just the examples above.

Your grammar can assume that unnecessary whitespace is automatically ignored. You don't need to write `@skip whitespace` below.

Your grammar can also assume that the expression has no unnecessary parentheses.

The first line of your grammar should be the rule for the root nonterminal.

Your answer should fit in the box without scrolling.

```
expr ::= union | disk ;
union ::= 'union' '(' expr ',' expr ')' ;
disk ::= 'disk' '(' 'new' 'Point' '(' coord ',' coord ')', ',' '1' ')' ;
coord ::= '0' | '-'?[1-9][0-9]* ;
```