CS 237 **Discrete Mathematics I**

Coolness

Predicates and Quantifiers *Exploration*

Goal

The purpose of this exploration is for you to *creatively* explore the cool meaning of *binary* predicates and *nested* quantifiers.

Requirements

Using the supplied `pq_sample.cpp` code as an example, and the `predicate.h` file (also supplied), create at least one **Predicate** subclass, and in your own `pq.cpp` file, implement these four **Predicate** functions (in addition to your subclass(es)):

1. `forAllForAll(int setX[], int sizeX, int setY[], int sizeY);` $\forall x \forall y\, P(x, y)$
2. `forAllForSome(int setX[], int sizeX, int setY[], int sizeY);` $\forall x \exists y\, P(x, y)$
3. `forSomeForAll(int setX[], int sizeX, int setY[], int sizeY);` $\exists x \forall y\, P(x, y)$
4. `forSomeForSome(int setX[], int sizeX, int setY[], int sizeY);` $\exists x \exists y\, P(x, y)$

As Rosen describes on pages 51-52 (THINKING OF QUANTIFICATION AS LOOPS):

> In working with quantifications of more than one variable, it is sometimes helpful to think in terms of nested loops. (Of course, if there are infinitely many elements in the domain of some variable, we cannot actually loop through all values. Nevertheless, this way of thinking is helpful in understanding nested quantifiers.) For example, to see whether $\forall x \forall y\, P(x, y)$ is true, we loop through the values for x, and for each x we loop through the values for y. If we find that $P(x, y)$ is true for all values for x and y, we have determined that $\forall x \forall y\, P(x, y)$ is true. If we ever hit a value x for which we hit a value y for which $P(x, y)$ is false, we have shown that $\forall x \forall y\, P(x, y)$ is false.
>
> Similarly, to determine whether $\forall x \exists y\, P(x, y)$ is true, we loop through the values for x. For each x we loop through the values for y until we find a y for which $P(x, y)$ is true. If for every x we hit such a y, then $\forall x \exists y\, P(x, y)$ is true; if for some x we never hit such a y, then $\forall x \exists y\, P(x, y)$ is false.
>
> To see whether $\exists x \forall y\, P(x, y)$ is true, we loop through the values for x until we find an x for which $P(x, y)$ is always true when we loop through all values for y. Once we find such an x, we know that $\exists x \forall y\, P(x, y)$ is true. If we never hit such an x, then we know that $\exists x \forall y\, P(x, y)$ is false.
>
> Finally, to see whether $\exists x \exists y\, P(x, y)$ is true, we loop through the values for x, where for each x we loop through the values for y until we hit an x for which we hit a y for which $P(x, y)$ is true. The statement $\exists x \exists y\, P(x, y)$ is false only if we never hit an x for which we hit a y such that $P(x, y)$ is true.

Test your implementation of these four functions from your own **main** function using a suitable binary (two-argument, or 2-ary) **Predicate** implementation instance. An example of such a **Predicate** is **GreaterThan**, whose `isTrue` method looks like this:

```
bool isTrue(int x, int y)
{
   return (x > y);
}
```

Note that this **Predicate** function has two arguments, `x` and `y`. Your creative task is to come up with your own binary **Predicate**, preferably more than one.

Export your write-up as a .pdf file named **cool.pdf** (making sure it IS a PDF file) and put it in the same folder as your code in your Linux account.

If conditions are right, you can build and submit your code in the Linux Lab via the command:

```
make it so
```

Comments

For your consideration, here are some comments former students have made about this exploration:

- The exploration was a great activity. I liked the challenge of figuring out how to do nested loops to accomplish the four functions. I also enjoyed looking for a predicate that could potentially be useful. I figured finding out if a point was on a line could be useful for collision detection in a game, for example. I also liked the exploration because I learned through trial and error how to check the coordinates with the line, and how to give better output.
- I think that the instructions on this were more than a little vague. I enjoy programming and I enjoy challenging myself, but when I don't know how it is that I am supposed to be challenging myself in an assignment, then how can I do so? I have worked through some obscure directions before, and it is nice to finally figure out how it is supposed to be done. But, with this assignment I just couldn't figure out if what I thought the instructions were saying was what they were really saying. If I did do this assignment correctly, then it really isn't anything difficult to do. Using logical operations in C++ to perform the things we are talking about in class does help me to better understand them. And the sample code did help me to reach the conclusion I drew from the instructions. So, for me the sample code did help and I was glad to have it as a reference.
- It's kind of fun to write a program that is much more about the math/logic than about the programing. It's also fun to have a math assignment that includes programming! I also liked the structure of the program. For example, if the four nested scope functions were implemented correctly, you could throw any predicate you want at them and know that that part of the logic will be correct. That way you can focus on the predicate itself. Yay for polymorphism! I wish I would have had more time to work on this. I would have written 7 or 8 predicates and tested them all!
- I liked being able to use my knowledge of Discrete Mathematics in a programming environment, and reinforcing the concepts by using them in this manner. I also liked having to use creativity to find solutions.
- I really like how I had to think about a domain or set to make each function evaluate to true and false. And this exploration was a good way to burn into my brain forAllforAll, forAllforSome, forSomeforAll, and forSomeforSome.

Grading Criteria

The rubric below is meant to guide you in your quest for exceptional quality.

	Exceptional 100%	**Good 90%**	**Acceptable 70%**	**Developing 50%**	**Missing 0%**
Application — using what you've learned 30%	Achieved new knowledge and figured out how it applies. At least ten elements of a good write-up are present.	Achieved new knowledge and figured out how it applies. At least nine elements of a good write-up are present.	Achieved new knowledge and figured out how it applies. At least seven elements of a good write-up are present.	Achieved new knowledge and figured out how it applies. At least five elements of a good write-up are present.	No application of anything learned. Zero elements of a good write-up are present (M.I.A., in other words).
Correctness/ Completeness 20%	Code compiles and works as expected. All functions work as required, **as shown by THOROUGH testing**.	Code compiles and mostly works as expected. At most three of the four functions works as required.	Code compiles and mostly works as expected. At most two of the four functions works as required.	Code compiles but does not work as expected. At most one of the four functions works as required.	Code does not even compile.
Creativity 40%	**Good**, plus the Predicates(s) is/are relatable to the everyday world, or something in your domain of discourse.	**Acceptable**, plus the Predicate uses numbers as indices to find non-numeric elements.	Created own predicates using numbers and arithmetic, or numbers related by a compound relational expression.	In addition to the **GreaterThan** example, used one very similar, such as **LessThan**.	Showed no creativity whatsoever. (Just used the **GreaterThan** example.)
Elegance 10%	**Good**, plus code uses helper functions to increase cohesion and minimize the complexity of the logic.	**Acceptable**, plus code is added to extend without modifying the given Predicate behaviors.	Code is correct, and avoids changing the signatures of the four functions.	Code is correct, as first and foremost, an elegant solution is a correct solution.	No elegance whatsoever. The code is not even correct.

What does **THOROUGH testing** mean? It means your test code exercises **both the true and the false return values** of each function, and your output is formatted such that it is **crystal clear** how all four functions' *expected* return values match their *actual* return values.