Comp 524 - Assignment 3: Social Distancing: Prolog

Date Assigned: September 18, 2021

Early Completion Date, Friday, Oct 1, 2021 (+5% extra credit)

Completion Date, Tuesday October 5, 2021

This assignment has two parts. The first one acquaints you with the Prolog interpreter and how to run local tests on Prolog code you have written. The second part involves writing the social distance application in Prolog.

Reference Material

Installing SWI-Prolog
Eclipse Prolog Plugin
SML and Prolog Installation and Use
https://www.cs.toronto.edu/~bonner/courses/2016f/csc324/handouts/usingSWIprolog.txt
https://www.swi-prolog.org/man/quickstart.html

Task 1: Install Prolog

Install a Prolog interpreter on your computer based on the reference material above.

Task 2: Prolog Greeting

Create a Prolog project/folder and to it the Prolog file named **Greeting.pl**. In the case of Java, you had flexibility in how we named the class and hence the file that contained the greeting code. This is because the class registry told the grader the name of the file. There is no equivalent of a class registry in other languages, so **it is important to use the file name required. Otherwise our grader will give you no points.**

Add the following text to this file, which defines a Prolog `greeting` rule:

```
greeting() :-
        write('hello\n'), write('goodbye\n').
```

Ask the interpreter to load this file using the command line, ProDT or some other system.

Once the file has been loaded, enter the following two lines on the console to "invoke" the rule and then exit the interpreter:

```
greeting().
halt.
```

Verify that strings **hello** and **goodbye** are output in response to the rule invocation.

Take a screenshot of this console input and output and put it in the Prolog project. You will also add screen shots of the more complex tasks below.

Task 3: Prolog LocalChecks (Optional)

Verify that you can execute **swipl** from the command line, that is, it is in your PATH.

LocalChecks are written in Java, so it is not possible to run the ones for Prolog from our Prolog project. This will be the case for other non-Java languages in which we code, Lisp and SML. So we will create a Java runner and tell it the name of our Prolog project location.

In an existing or new java project, create a Java class whose contents are of the form shown below:

```
import gradingTools.comp524f21.assignment3.F21Assignment3Suite;
import trace.grader.basics.GraderBasicsTraceUtility;
import util.trace.Tracer;
public class RunF21A2Tests {
	static final String PROJECT_LOCATION =
"D:/dewan_backup/Java/PLProjs/PLProjsProlog"; //Your location will be different

	public static void main (String[] args) {
//		Tracer.showInfo(true);
//		GraderBasicsTraceUtility.setBufferTracedMessages(false);
		F21Assignment3Suite.setProjectLocation(PROJECT_LOCATION);
		F21Assignment3Suite.main(args);
	}
}
```

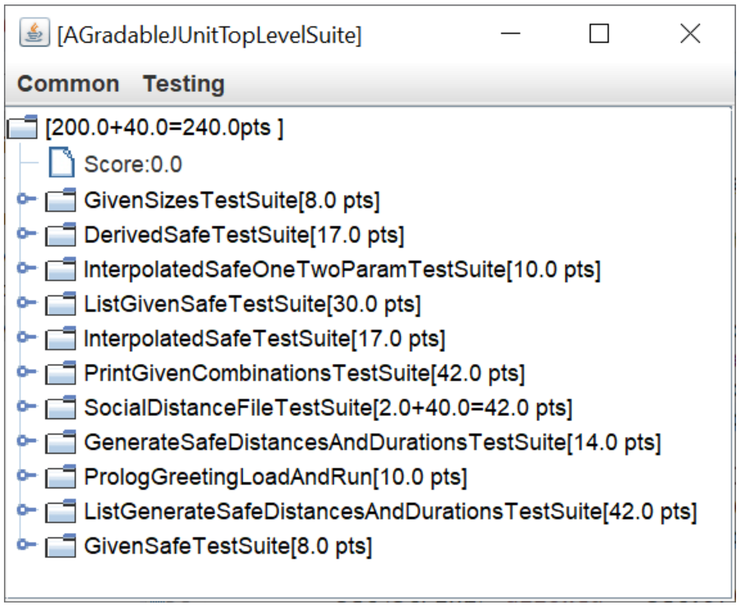
Right click on your Prolog project. Go to Properties Resource. Copy the folder name after **Location**: Assign it to the PROJECT_LOCATION string constant above.

The call:

```
F21Assignment3Suite.setProjectLocation(PROJECT_LOCATION);
```

tells LocalChecks that the project to be graded is not the one from which this runner class is called. Instead, it is, `PROJECT_LOCATION.`

If you now execute this runner, you should see the checks for the Prolog project and not the Java project from which you ran this class. Double-clicking on the *PrintGreetingLoadAndRun* test in this suite should make it run successfully.



```
[AGradableJUnitTopLevelSuite]
Common   Testing
[200.0+40.0=240.0pts ]
    Score:0.0
    GivenSizesTestSuite[8.0 pts]
    DerivedSafeTestSuite[17.0 pts]
    InterpolatedSafeOneTwoParamTestSuite[10.0 pts]
    ListGivenSafeTestSuite[30.0 pts]
    InterpolatedSafeTestSuite[17.0 pts]
    PrintGivenCombinationsTestSuite[42.0 pts]
    SocialDistanceFileTestSuite[2.0+40.0=42.0 pts]
    GenerateSafeDistancesAndDurationsTestSuite[14.0 pts]
    PrologGreetingLoadAndRun[10.0 pts]
    ListGenerateSafeDistancesAndDurationsTestSuite[42.0 pts]
    GivenSafeTestSuite[8.0 pts]
```

The Mac implementation of Java and Eclipse seems to be inconsistent with the specifications. So you might get a NotGradableException. If so, do the following.

On the command line, execute, which swipl.

```
$ which swipl
/d/Program Files/swipl/bin/swipl
```

Here, /d/Program Files/swipl/bin/swipl, is the **full name** of swipl.
Import the following class in the runner:

import grader.basics.execution.prolog.PrologCommandGeneratorSelector;

Now in your runner main method, before you execute the main method of the suite execute the following call, passing to setUserBinary the full name of **swipl**:

```
        PrologCommandGeneratorSelector.getCommandGenerator().
                setUserBinary(<full name of swipl >);
```

For example, based on the result of my which command, I would make the following call:

```
PrologCommandGeneratorSelector.getCommandGenerator().
                setUserBinary(“/d/Program Files/swipl/bin/swipl”);
```

Task 4: Social Distancing without Lists

Now comes the more interesting part, wherein you will design and implement your own code.

You are restricted to Prolog constructs identified in this course – the use of some of Prolog's imperative and built-in features defeats the purpose of the assignment.

You will implement the application with and without lists and recursion.

You are required to create code that is correct and follows style rules. As you are new to Prolog, focus first on correctness **if style hints are confusing and style requirements are hard to follow**. Next try to follow the rules. The style hints may, in fact, allow you to converge to the solution quicker. So make decisions based on each style hint and requirement.

As you did in in the previous part, you will create in your project directory a file whose name is given to you. For this and the next part, it will be: **SocialDistance.pl.** We will refer to this as the specification file as it will contain the rules and facts you have specified.

You should use tracing to debug your queries. Tracing can be turned or off by executing **trace()** and **notrace(),** respectively**.** A nice description of tracing and notracing and other user-interface features of swipl is in:
https://www.cs.toronto.edu/~bonner/courses/2016f/csc324/handouts/usingSWIprolog.txt

You may want to also look at swipl quick guide:
https://www.swi-prolog.org/man/quickstart.html

You can search the web for other material.

You can debug code using also the **write**() query you used in part 1. Make sure you remove or comment out calls to this query when you submit the code.

Prolog Syntax Gotchas

Here are some mistakes I tend to make based on the differences between popular imperative languages and Prolog:

1. Using a semicolon instead of a period to end a fact or rule. A Prolog semicolon is analogous to an imperative **or** operation.
2. Putting one or more spaces between a relation name and the parenthesized list of parameters, e.g. typing givenSafe (Distance, Duration, Exhalation) instead of givenSafe(Distance, Duration, Exhalation).
3. Using <= instead of =<.
4. Failing to capitalize variables - may lead to confusion with queries simply returning false

Please expand on this list as you find your own gotchas.

As you see here, we are omitting the prefix "is" before a query name. Thus, it is givenSafe instead of isGivenSafe. This is because all Prolog queries are essentially Boolean functions when their parameters are constants, and some of them return non Boolean results when executed successfully with variable parameters.

givenSizes

Recall this table from A1, giving values to the small, medium and large values for each of our features.

	Small	Medium	Large
Distance	6	13	27
Duration	15	30	120
Exhalation Level	10	30	50

Table 1 Small, Medium and Large Values

Add facts and/or rules (to the specification file) that support a query named **givenSizes** taking as int parameters Distance, Duration, and Exhalation Level (in that order). Here is its behavior:

```
:-givenSizes(6, 15, 10).
true :-;
false.

|givenSizes(6, 13, 27).
false.

:-givenSizes(Distance, 15, 10).
Distance = 6 :-;
false.

:-givenSizes(Distance, Duration, Exhalation).
Distance = 6,
Duration = 15,
Exhalation = 10 :-;
Distance = 13,
Duration = Exhalation, Exhalation = 30 |;
Distance = 27,
Duration = 120,
Exhalation = 50.
```

When the parameters are all constants, it determines if the **column** represented by the parameter s exists in the table. When a parameter is a variable, it lists the small, medium and/or large value matching the other parameters, if such a match exists. When all parameters are variables, it prints the whole table.

Please check all examples for errors!

Stylistic hints, which may also help you in correctness.

Consider the smaller problems of:
(a) Verifying if a value occurs in a specific cell specified by a query name.
For example, does the value occur in cell (0, 0), that is, is it a small distance. There are nine cells, so you can support a differently named query for each of these cells

(b) Verifying if a tuple occurs in a specific column specified by a query name. For example, does a tuple occur in Column 0, that is, does it contain a small distance, duration and exhalation level. You can support a differently named query for each column.

Ignore these hints if they are confusing.

givenSafe

Consider again the following table containing the safe combinations.

Distance	**Duration**	**Exhalation Level**
Medium	Medium	Medium
Small	Medium	Small
Large	Medium	Large
Medium	Small	Large
Medium	Large	Small
Large	Large	Medium
Small	Small	Medium

Table 2 Assumed Given Safe Combinations

Support a query named **givenSafe** that takes as parameters the Distance, Duration, and Exhalation Level. Here is the behavior:

```
|givenSafe(13, 30, 30).
true :-;
false.

|givenSafe(13, 29, 29).
false.

|givenSafe(13, Duration, Exhalation).
Duration = Exhalation, Exhalation = 30 :-;
Duration = 15,
Exhalation = 50 |;
Duration = 120,
Exhalation = 10 |;
false


|givenSafe(Distance, Duration, Exhalation).
Distance = 13,
Duration = Exhalation, Exhalation = 30 :-;
Distance = 6,
Duration = 30,
Exhalation = 10 |;
Distance = 27,
Duration = 30,
Exhalation = 50 |.
```

The input period after the partial results shows that continuations can be stopped by entering this period.

When the parameters are all constants, the query determines if the **row** represented by the parameters exists in Table 2.

When one or more parameters are variables, the query lists the values of the variables that match each other and the constant parameters, if such matches exists, and returns false otherwise.

When all parameters are all variables, it prints the whole table.

Again, please check examples for errors!

derivedSafe

Support a query named **derivedSafe** that takes as parameters the Distance, Duration, and Exhalation Level. When all parameters are constants, it returns the same Boolean value as your corresponding Java function call, isDerivedSafe() would. Because of the properties of relational operators in Prolog, this query cannot be called with variable parameters. Here is the illustrated behavior.

```
|:-derivedSafe(13, 30, 30).
true :-;
false.

|derivedSafe(13, 29, 29).
true :-;
false.

|derivedSafe(13, 31, 31).
false.

:-derivedSafe(13, Duration, Exhalation).
```

```
ERROR: Arguments are not sufficiently instantiated
ERROR: In:
ERROR:    [9] _5202=<30
ERROR:    [8] derivedSafe(13,_5230,_5232) at
```

Style Hint: *Consider the three smaller problems of determining if a distance, duration, or exhalation level is at least as safe a given distance, duration and exhalation level.*

We know, for example, that a distance is at least as safe as another if it is greater than or equal to that distance. Each of these problems can be associated with its own two parameter query.

Style Rules

1. *Mnemonic components in camel case identifier names.*
2. *No repetition* of strings or numbers.
3. *No magic numbers*.
4. *Separation of concerns*, which implies you should decompose rules into sub-rules, thereby also supporting more queries.
5. *Document hint-based sub rules*: % before a line is a comment. If you have created a non-required rule based on some hint given here, put *all italicized* sentences describing the hint, verbatim, as a comment, immediately before the sub-rule. You can have the same hint associated with multiple sub rules.

Three-Parameter interpolatedSafe

Support a query named **interpolatedSafe** that takes as parameters the Distance, Duration, and Exhalation Level. When all parameters are constants, it returns the same Boolean value as your corresponding three-parameter Java function call, isInterpolatedSafe() would.

Remember the alternative to using conditionals in Prolog is to define a Prolog rule for each case that checks the condition of the case.

Because of the properties of relational operators in Prolog, this query also cannot be called with variable parameters:

```
|:-interpolatedSafe(13, 30, 30).
true :-;
false.

|interpolatedSafe(13, 29, 29).
true :-;
false.

|interpolatedSafe(100, 29, 29).
false.

:-interpolatedSafe(Distance, 29, 29).
```

```
ERROR: Arguments are not sufficiently instantiated
ERROR: In:
```

I do not know of a way to get a value of maximum integer in Prolog, so use the arbitrary value of 200 in the definition of high interpolation.

Two-Parameter interpolatedSafe

Support a query named **interpolatedSafe** that takes as parameters the Distance and Duration. When all parameters are constants, it returns the same Boolean value as your corresponding two-parameter Java function call, isInterpolatedSafe() would.

```
:-interpolatedSafe(13, 30).
true :-;
false.

|interpolatedSafe(13, 29).
true :-;
false.
```

One-Parameter interpolatedSafe

Support a query named **interpolatedSafe** that takes as parameters the Distance. When the parameter is a constant, it returns the same Boolean value as your corresponding one-parameter Java function call, isInterpolatedSafe() would.

```
|interpolatedSafe(13).
true :-;
false.
|interpolatedSafe(12).
false.
:-interpolatedSafe(14).
true :-;
false.
```

generateSafeDistancesAndDurations

This query is meant to take variables for Distance and Duration and a constant for the exhalation level. Each result of the query assigns to the two variables a distance and duration value in Table 2, respectively, associated with an interpolation of the exhalation level.

The query below interpolates the exhalation level to 10 and then prints the values of Distance and Duration in Table 2 that match it.

```
|:-generateSafeDistancesAndDurations(Distance, Duration, 9).
Distance = 6,
Duration = 30 :-;
Distance = 13,
Duration = 120 |;
```

```
false.
```

The query below interpolates the exhalation level to 30 and then prints the values of Distance and Duration in Table 2 that match it.

```
|generateSafeDistancesAndDurations(Distance, Duration, 11).
Distance = 13,
Duration = 30 :-;
Distance = 27,
Duration = 120 |;
Distance = 6,
Duration = 15 |;
false.
```

The following query interpolates the exhalation level to infinity, cannot find any matching tuple in the table, and prints false.

```
|generateSafeDistancesAndDurations(Distance, Duration, 51).
false.
```

If the first two arguments are constants, then the query simply returns a Boolean indicating whether their combination with the interpolated value of the exhalation level is safe.

Part 3: List-based queries

Define Table 2 now also as a nested tuple-list.

The remainder of the assignment requires you to use recursion to traverse this list.

listGivenSafe

This is like the givenSafe query you implemented above without lists except that it takes as an argument a single tuple. The first, second and third elements of the tuple correspond to the three arguments of the givenSafe query, as illustrated below.

```
|listGivenSafe([13, 30, 30]).
true :-;
false.

|listGivenSafe([13, 29, 29]).
false.

:-listGivenSafe([13, Duration, Exhalation]).
Duration = Exhalation, Exhalation = 30 :-;
Duration = 15,
Exhalation = 50 |;
Duration = 120,
Exhalation = 10 |;
false.
```

```
|listGivenSafe(SafeTuple).
SafeTuple = [13, 30, 30] :-;
SafeTuple = [6, 30, 10] |;
SafeTuple = [27, 30, 50] |;
SafeTuple = [13, 15, 50] |;
SafeTuple = [13, 120, 10] |;
SafeTuple = [27, 120, 30] |;
SafeTuple = [6, 15, 30] |;
false.
```

Hint: Call another query to create the full safety table, and then recursively process the table.

printGivenCombinations

This is similar to the corresponding Java method you implemented, except that it only prints given combinations (it does not print generated random combinations). It uses the same output format as the Java method.

To give you practice with Prolog arithmetic, this query takes an int argument, N. It prints the first N elements of the table, if N is greater than the size of the table. Otherwise it prints the entire table. The header is always printed.

```
:-printGivenCombinations(0).
Distance, Duration, Exhalation, IsSafe
false.
:-printGivenCombinations(3).
Distance, Duration,  Exhalation, IsSafe
13,30,30,true
6, 30,10,true
27,30,50,true
false.
:-printGivenCombinations(7).
Duration, Distance, Exhalation, IsSafe
30,13,30,true
30,6,10,true
30,27,50,true
15,13,50,true
120,13,10,true
120,27,30,true
15,6,30,true
false.

 :-printGivenCombinations(10).
Duration, Distance, Exhalation, IsSafe
30,13,30,true
30,6,10,true
30,27,50,true
15,13,50,true
120,13,10,true
```

```
120,27,30,true
15,6,30,true
false.
```

Hint: You will need to recursively reduce both the safety table and N.

listGenerateSafeDistancesAndDurations

This is like the generateSafeDistancesAndDurations query you implemented above without lists except that it takes different arguments. The exhalation level is now the first argument. The second argument is a list of (Distance, Duration) pairs.

If these pairs are constants the query returns a Boolean answer indicating if each pair along with the interpolated value of the first argument is safe tuple.

If these pairs are variables, it generates a list of distances and durations that together with the interpolated value of the first argument, form safe tuples in Table 2.

```
:-listGenerateSafeDistancesAndDurations(9, GeneratedTable).
GeneratedTable = [[6, 30], [13, 120]] :-;
false.
```

```
:-listGenerateSafeDistancesAndDurations(11, GeneratedTable).
GeneratedTable = [[13, 30], [27, 120], [6, 15]] :-;
false.
```

```
|listGenerateSafeDistancesAndDurations(51, GeneratedTable).
GeneratedTable = [] :-;
false.
```

The query will need to recursively both reduce the safety table and append to the result table.

Submission Material

As always submit the entire project folder to both **Sakai** and **Gradescope;** include **png** files with **localchecks** result; and include **png** one or more screenshots demonstrating the functionality (queries) you have implemented.