1) Consider a sequence of integers formed in the following manner:

Start the sequence with any positive integer, creating an initial sequence with length 1
While the last term in the sequence is greater than 1
    Let n represent the last number in the sequence
    If n is even then
        Concatenate n divided by 2 to the end of the sequence
    Else if n is odd then
        Concatenate 3 * n + 1 to the end of the sequence

It is theorized (but has never been proved) that a sequence constructed in this manner will always terminate with a 1 when it begins with a positive integer.

[3 marks] Write a Haskell function named getSequence that takes an integer as its only parameter, which is the starting number for the sequence. The function should return the list of integers for a sequence constructed using the previous algorithm as its only result.

Test cases to consider:

```
getSequence 1 should return [1]
getSequence 2 should return [2,1]
getSequence 3 should return [3,10,5,16,8,4,2,1]
getSequence 4 should return [4,2,1]
getSequence 5 should return [5,16,8,4,2,1]
getSequence 104 should return [104,52,26,13,40,20,10,5,16,8,4,2,1]
```

[4 marks] Continue your work on this problem by creating a second Haskell function named findSequences. This function will take two integer parameters. The first integer is the number, N, of sequences that need to be located. The second integer is the length, L, of the sequences that need to be located. The function will return a list of N integers which are the initial integers of the first N sequences that have length L. Note that your function should work for any positive integer N and any positive integer L (of course, some values will take a very long time compute). Do not impose an artificial limit of the number or length of the sequences as doing so will result in a loss of 2 marks on this part of the question.

Test cases to consider:

```
findSequences 1 1 should return [1]
findSequences 1 101 should return [107]
findSequences 2 101 should return [107,644]
findSequences 3 10 should return [12,13,80]
findSequences 3 256 should return [20830,20831,20895]
findSequences 8 100 should return [322,323,1788,1789,1790,1791,1936,1940]
findSequences 11 15 should return [11,68,69,70,75,384,416,424,426,452,453]
```

2) **[7 marks]** Consider a list of (String, String) pairs where the first element in each pair represents a predator and the second element in each pair represents a prey. For example, the pair ("Lion", "Zebra") indicates that lions eat zebras while the pair ("Zebra", "Grass") indicates that zebras eat grass. A list of such pairs can be used to describe the predator-prey relationships within an ecosystem.

Write a Haskell function named getPreyLists and transforms a list of predator-prey lists into a list of tuples where the first element in each tuple is the name of the predator, and the second element in each tuple is a list of all of the elements preyed on by that predator. For example:

```
getPreyLists [("Lion", "Zebra"), ("Lion", "Gazelle"), ("Zebra", "Grass"),
("Zebra", "Bush"), ("Gazelle", "Bush")]
```

should return

```
[("Lion",["Zebra","Gazelle"]),("Zebra",["Grass","Bush"]),("Gazelle",["Bush"])]
```

Note that you cannot assume that all occurrences of a predator will be grouped together in the list of (predator, prey) tuples. Predators should appear in the newly constructed list in the same order in which they were first encountered in the original list, and similarly, prey should occur in the result list in the same order in which they occurred in the original list for each predator.

Your implementation of getPreyList must consist of only one line (plus its type signature), and that line must use one of Haskell's fold functions to fold a helper function (that you will also write) over the list to compute the desired value. A solution that fails to use fold in a reasonable way will not receive any credit. Hint: I solved the problem with a left fold and recommend that you do the same.

Test cases to consider:

`getPreyLists []` should return `[]`

`getPreyLists [("7", "9")]` should return `[("7",["9"])]`

`getPreyLists [("a","b"),("a","c")]` should return `[("a",["b","c"])]`

`getPreyLists [("Fox", "Toad"), ("Toad", "Insect"), ("Insect", "Plant")]`
should return `[("Fox",["Toad"]),("Toad",["Insect"]),("Insect",["Plant"])]`

`getPreyLists [("A","B"),("A","C"),("A","D"),("A","E")]` should return
`[("A",["B","C","D","E"])]`

`getPreyLists [("A","B"),("A","C"),("A","D"),("B","C"),("B","D"),("C","E"),("D","E")]`
should return `[("A",["B","C","D"]),("B",["C","D"]),("C",["E"]),("D",["E"])]`

`getPreyLists [("D","E"),("A","B"),("A","D"),("B","D"),("C","E"),("B","C"),("A","C")]`
should return `[("D",["E"]),("A",["B","D","C"]),("B",["D","C"]),("C",["E"])]`

3) Geologic time is broken down into 4 eons:

Eon	Years
Hadean	4,600,000,000 years ago to less than 4,000,000,000 years ago
Archean	4,000,000,000 years ago to less than 2,500,000,000 years ago
Proterozoic	2,500,000,000 years ago to less than 541,000,000 years ago
Phanerozoic	541,000,000 years ago to present

[2 marks] Create an enumerated type named Eon for the eons listed in the table above. Each of the eons named previously should be a constructor for your type. Your enumerated type should allow eons to be tested for equality (each eon is equal to itself and nothing else), ordering (older eons are less than more recent eons), and allow a list of eons to be constructed with the .. operator.

Test cases to consider:

```
Hadean == Archean should return False
Proterozoic == Proterozoic should return True
Archean < Phanerozoic should return True
Archean < Hadean should return False
length [Phanerozoic, Proterozoic, Hadean] should return 3
length [Proterozoic .. Hadean] should return 0
```

[2 marks] Extend your type so that it also allows eons to be converted to strings. When an eon is converted to a string it should include the name of the eon, along with a description of it's time period in parentheses. The time periods should be expressed as millions / billions of years ago. For example:

```
show Hadean should return "Hadean (4.6 billion to 4.0 billion years ago)"
show Archean should return "Archean (4.0 billion to 2.5 billion years ago)"
show Proterozoic should return
  "Proterozoic (2.5 billion to 541 million years ago)"
show Phanerozoic should return
  "Phanerozoic (541 million years ago to present)"
```

[3 marks] Write a function named yearToEon which takes an integer year as its only parameter and returns an eon as its result. Your function should report an error for parameter values greater than 2017 or less than -4,600,000,000.

Test cases to consider:

```
yearToEon (-1) should return Phanerozoic
yearToEon (-4000000000) should return Archean
```

Note that when you run your function interactively it will display the eon with its description if you have completed the second part of this question rather than only displaying the eon name.

4) Consider a knowledgebase full of facts that describe predators and the prey that they eat. Each fact in the knowledgebase will be named eats and will take 2 parameters. The first parameter is the name of a predator and the second is the name of an animal that the predator preys on. A collection of such facts describes a food web for an ecosystem. Such collections of facts can be found on the course website in Aquatic.pro and Simple.Pro and Mixed.pro.

[2 marks] A producer is an organism in the food web that is consumed by one or more organisms and does not consume any other organisms itself. Write a Prolog rule named isProducer that takes 1 parameter. That parameter will always be an atom. The rule should succeed if that atom is a producer in the food web. Otherwise is should fail. Take whatever steps are necessary to ensure that only one copy of the result is generated. (Additional copies of the same answer should **not** be generated via backtracking).

Test cases to consider (based on the facts in Aquatic.Pro):

isProducer(phytoplankton) should succeed.
isProducer(seaweed) should succeed.
isProducer(bird) should fail.

[5 marks] The height of an organism within the food web is the length of the longest path from the organism to any producer. Write a Prolog rule named height that takes two parameters. The first parameter will always be an atom that is the name of an organism in the food web. The second parameter will always be an uninstantiated variable. The rule should instantiate the variable to the height of the organism provided as the first parameter. Take whatever steps are necessary to ensure that only one copy of the result is generated. (Additional copies of the same answer should **not** be generated via backtracking).

Hint: I found it helpful to a couple of Prolog's build-in list processing predicates when completing this problem (among other predicates). The list processing predicates are described in Section 8.20 of the gProlog manual which is available online.

Test cases to consider (based on the facts in Aquatic.Pro):

height(phytoplankton, X) should instantiate X to 0 and succeed.
height(zooplankton, X) should instantiated X to 1 and succeed.
height(mussels, X) should instantiate X to 2 and succeed.
height(crab, X) should instantiate X to 3 and succeed.
height(bird, X) should instantiate X to 4 and succeed.

The code that you write should not generate any warnings.

5) **[7 marks]** Consider a knowledgebase that includes facts about flights between airports. Such a knowledgebase can be found on the course website as Flights.pro. Each fact describes a flight from one airport to another, along with when the flight departs (some number of minutes after midnight), and its duration (in minutes).

The following rule determines whether or not there is a flight between two airports departing at or after specific point in time (described as a number of minutes after midnight). Whenever the rule is used the first three parameters will be instantiated to two cities and a number of minutes. The last three parameters will always be uninstantiated variables that will be instantiated to the actual departure time of the flight (number of minutes after midnight), the arrival time (number of minutes after midnight) and the route taken (a list containing of a single fly structure from the departure city to the arrival city).

```
routeAfter(C1, C2, SoonestAllowed, DepartureTime, ArrivalTime, Route) :-
  flight(C1, C2, DepartureTime, Duration),
  DepartureTime >= SoonestAllowed,
  ArrivalTime is DepartureTime + Duration,
  Route = [fly(C1, C2)].
```

For example, the goal `routeAfter(atl, cae, 0, Dep, Arr, Route)` generates the following results:

```
Arr = 1416
Dep = 1356
Route = [fly(atl,cae)] ? ;

Arr = 1294
Dep = 1235
Route = [fly(atl,cae)] ? ;

Arr = 966
Dep = 908
Route = [fly(atl,cae)] ? ;

Arr = 610
Dep = 556
Route = [fly(atl,cae)] ? ;
```

While the rule above handles direct flights correctly, it doesn't consider connecting flights. Your task is to write a second, recursive, rule for routeAfter that takes the same parameters as the first. The Route variable should be instantiated to a list that describes each flight segment as a flight from one city to another, and each layover with its duration. The flights will be stored in a fly structure just

like the ones used by the provided version of the rule. The layovers (the time spent at a connecting airport) will be described by a layover structure that contains only one value, which is the duration of the layover. **Layovers must be at least 60 minutes in length to be valid.** Any route that contains a layover of less than 60 minutes should **not** be included in the generated results. Similarly, the routes generated by this recursive rule should not contain any cases where the same city is visited twice (or more). For example, `routeAfter(dfw, jac, 0, Dep, Arr, Route)` should generate the following results:

```
Arr = 913
Dep = 360
Route = [fly(dfw,slc),layover(307),fly(slc,jac)] ? ;

Arr = 755
Dep = 360
Route = [fly(dfw,slc),layover(147),fly(slc,jac)] ? ;

Arr = 847
Dep = 345
Route = [fly(dfw,atl),layover(122),fly(atl,jac)] ? ;
```

Another example to consider: `routeAfter(mia, jac, 0, Dep, Arr, Route)` should generate the following results:

```
Arr = 913
Dep = 420
Route = [fly(mia,slc),layover(90),fly(slc,jac)] ? ;

Arr = 847
Dep = 330
Route = [fly(mia,atl),layover(150),fly(atl,jac)] ? ;
```

Another example to consider: `routeAfter(jac, mke, 0, Dep, Arr, Route)` should generate the following results:

```
Arr = 1290
Dep = 777
Route = [fly(jac,atl),layover(158),fly(atl,mke)] ? ;

Arr = 1484
Dep = 777
Route = [fly(jac,atl),layover(356),fly(atl,mke)] ? ;

Arr = 1282
```

```
Dep = 796
Route = [fly(jac,msp),layover(257),fly(msp,mke)] ? ;

Arr = 1484
Dep = 796
Route = [fly(jac,msp),layover(110),fly(msp,atl),layover(143),fly(atl,mke)]
? ;

Arr = 1484
Dep = 745
Route = [fly(jac,jfk),layover(64),fly(jfk,atl),layover(128),fly(atl,mke)]
? ;
```

Finally, `routeAfter(mke, jac, 0, Dep, Arr, Route)` should generate the following results:

```
Arr = 830
Dep = 395
Route = [fly(mke,msp),layover(188),fly(msp,jac)] ? ;

Arr = 913
Dep = 395
Route = [fly(mke,msp),layover(62),fly(msp,slc),layover(116),fly(slc,jac)]
? ;

Arr = 847
Dep = 350
Route = [fly(mke,atl),layover(119),fly(atl,jac)] ? ;
```

The code that you write should not generate any warnings.