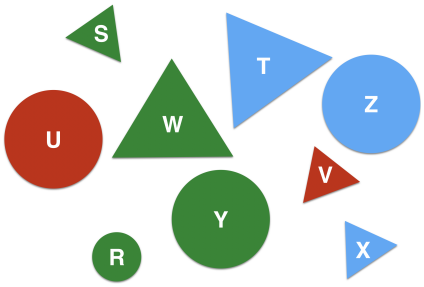
Graphs and Relations

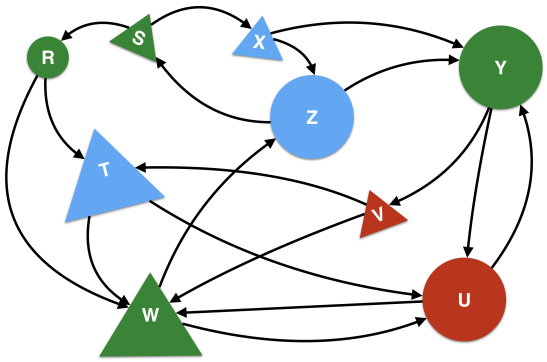
1 A Graph of Things

Remember the universe of discs and triangles from the first tutorial? We still have the universe




```
data Thing = R|S|T|U|V|W|X|Y|Z
things = [ R, S, T, U, V, W, X, Y, Z ]
```

and predicates, `isRed`, `isGreen`, `isBlue`, `isSmall`, `isBig`, `isDisc`, `isTriangle`. We add a relation to this universe. We say "X likes Y" if there is an arrow from X to Y. Here is the universe with the relation: We call this a directed graph.



This relation is defined for you in Tutorial5.hs[1], implemented as the function:

```
likes :: Thing -> Thing -> Bool -- x `likes` y iff there is an arrow x -> y
```

```
GHCi> likes R T
True
GHCi> likes R U
False
```

It is often clearer to use `` `likes` `` in infix style, with backquotes (`):

```
GHCi> R `likes` T
True
GHCi> R `likes` U
False
```

[1] https://gavinphr.github.io/INF1A/tutorial5/Tutorial5.hs

For each of the following statements, determine whether it is true by just looking at the graph and then use Haskell to verify your answer.

1. Some thing likes S.

   Solution: True.

   ```
   GHCi> or [x `likes` S | x <- things]
   True
   ```

2. Every thing likes W.

3. Every thing is liked by some thing.

4. Some thing is not liked by any thing.

   What is the relationship between statement 3 and 4?

2 Quantifiers and Formal Representation

Natural languages such as English and Chinese are intrinsically ambiguous. Statements in natural languages can be interpreted in many different ways. To avoid ambiguity (and for other benefits), let's try to formally represent the statements in the previous exercise.

First we need the two **quantifiers**:

1. forall $\forall$
2. exists $\exists$

So for the first statement, **"Something likes S."**, we write:

$$\exists \texttt{x} \in \texttt{things}\ (\texttt{x `likes` S})$$

It reads: there exists an `x` in `things` such that `x `likes` S`.

For the third statement, **"Everything is liked by something."**, we write:

$$\forall \texttt{x} \in \texttt{things}\ (\exists \texttt{y} \in \texttt{things}\ (\texttt{y `loves` x}))$$

Convince yourself that the formal statement conveys the same meaning as the original.

A benefit of formal statements is that we can easily turn them into Haskell code. We first need two auxiliary functions to represent ∀ and ∃:

```
every :: [t] -> (t -> Bool) -> Bool
every xs p = and [p x | x <- xs]

some :: [t] -> (t -> Bool) -> Bool
some ys p = or [p y | y <- ys]
```

You will often find it useful to use a λ-functions or section as the predicate, `p`.

For example, ∃`y` ∈ `things` (`y` `` `likes` `` `S`) may be translated to:

```
some things (\y -> y `likes` S)
```

```
GHCi> some things (\y -> y `likes` S)
True
GHCi> some things (`likes` S) -- we can also use a section
True
```

And ∀`x` ∈ `things` (∃`y` ∈ `things` (`y` `` `loves` `` `x`)) is translated to:

```
every things (\x -> some things (\y -> y `likes` x))
every things (\x -> some things (`likes` x)) -- again we can use a section
```

```
GHCi> every things (\x -> some things (`likes` x))
True
```

Translate the rest of the statements, first to formal mathematical representations, then to Haskell.

5. Every thing likes W.

6. Some thing is not liked by any thing.

3 A Social Graph

In the first exercise, we could answer all of the questions by just looking at the graph. In this exercise, the graph is much larger. We have 100 people, given as a list of type `[Person]` :

```
data Person = Carmina|Malika|Margherita|Ginny|Loida|Rosalie|Nikki
            -- the list continues ...

people :: [Person]
people = [Carmina, Malika, Margherita, Ginny, Loida, Rosalie, Nikki,
    Bea, Delores, Soledad, Sheba, Elsy, Lashanda, Carolynn, ......
```

Each person can be:

- either happy or not happy.
- either a scientist or not a scientist.
- either an artist or not an artist.
- either a politician or not a politician.

You can check whether a person has a particular property using the predicates:

```
isHappy     :: Person -> Bool          isArtist     :: Person -> Bool
isScientist :: Person -> Bool          isPolitician :: Person -> Bool
```

For example, we can find out whether Carmina is happy:

```
GHCi> isHappy Carmina
False
```

:-(. Observe that one can have more than 1 property i.e. a person can be happy, a scientist, an artist, and a politician at the same time.

There is a *loves* relation in our social graph that is similar to the *likes* relation in the first exercise. We can check whether Carmina loves Brain using the function `loves :: Person -> Person -> Bool`:

```
GHCi> Carmina `loves` Brian
True
```

For each of the statements below, first turn it into a formal mathematical statement, then translate the formal statement into Haskell code and determine whether the statement is true. A couple of examples are already done for you.

7. Everybody loves somebody.

Solution: $\forall$`x` $\in$ `people` $\exists$`y` $\in$ `people` (`x` `` `loves` `` `y`)

```
every people (\x -> some people (\y -> x `loves` y))
```

8. Somebody loves everybody. (In English, this statement is ambiguous. Give a two formalisations, one for each possible meaning and check whether either is true in our universe.

9. Everybody loves someone who loves them.

10. Somebody loves someone who loves them.

11. Everybody (x), loves some y, who loves some z, who loves them (x).[2]

12. Every politician loves an artist.

Solution: ∀x ∈ Politicians (∃y ∈ Artists (x `loves` y))

```
every [x | x <- people, isPolitician x] (\x -> some
[y | y <- people, isArtist y] (\y -> x `loves` y))
```

13. Everyone loved by an artist loves a scientist.

14. Some artist has only one love.

    Note that a common trick for expressing the "only one" constraint is by decomposing it into two constraints: "at least one" and "no more than one".

[2] In other words, everyone belongs to some love triangle.

Six degrees of separation is the idea that all people are six, or fewer, social connections away from each other. It is claimed that, a chain of "a friend of a friend" statements can be made to connect any two people in a maximum of six steps.[3] In this exercise, you will find the degree of separation of our social graph, also called the *diameter* of the graph.

In our case, the social connection is the `loves` relation. Let's first think about what it means for a person `x` to be able to reach another person `z` in one step, it is simply that:

```
reach1 :: Person -> Person -> Bool
reach1 x z = x `loves` z
```

How can we make a function that tells us whether person `x` is able to reach person `z` in 2 steps? If there is a person `y` such that `x` loves (i.e. `x` can reach `y` in 1 step) and `y` can reach `z` in 1 step, then `x` can reach `z` in 2 steps.

How can we tell whether `x` can reach `z` in 3 steps? If there is a `y` such that `x` likes and `y` can reach `z` in 2 steps, then `x` can reach `z` in 3 steps.

15. Now implement the functions `reach2` and `reach3`.

16. What is the degree of separation in our graph?

[3] https://en.wikipedia.org/wiki/Six_degrees_of_separation

4 Sudoku

A site listing all known 17-clue sudoku problems with a unique solution encodes a sudoku as a string of 81 digits. Each character represent a square on the board. The first nine characters represent the first row, and so on; 0 represents a blank square, and a digit 1..9 represents an entry in the corresponding square. The file Tutorial5.hs includes three examples of proposed solutions, and a function

```
entries :: String -> Int -> Int -> Int -> Bool
entries str i j k
  | length str /= 81 || or [ not (isDigit c) | c <- str] = error "Ill-formed string"
  | or [ arg < 1 || arg > 9| arg <- [i,j,k] ] =  error "index out of range [1..9]"
  | otherwise = k > 0 && str!!(9 * (i - 1) + (j -1)) == intToDigit k
```

that converts such a string `prob` into a predicate `entered = entries prob`, taking three integer arguments in the range [0..9]) – `entered i j k` is `True` iff *the digit k is **entered** in square (i, j)*.

Your task is to complete the following function to check whether the examples really are solutions. Do they satify the rules of sudoku?

```
check :: (Int -> Int -> Int -> Bool) -> Bool
check entered =
  -- every square is filled
  and [ or[entered i j k | k <- [1..9] ]
      | i <- [1..9], j <- [1..9] ]
  && -- no square is filled twice
  and [ or[ not(entered i j k), not (entered i j k') ]
      | i <- [1..9], j <- [1..9], k <- [1..9], k' <- [1..(k-1)]]
  && -- every row contains every digit
  and [ or [entered i j k | j <- [1..9] ]
      | i <- [1..9], k<- [1..9] ]
  -- every column contains every digit
  -- every big square contains every digit
```

17. Complete the code to check for the two remaining properties, and check the three examples.