Useful Functions from the Haskell Standard Prelude

\[
\begin{align*}
\text{fst} \quad \text{snd} & \quad :: \ (a, b) \to a \\
\text{fst} \ (x, \_ ) & \quad = x \\
\text{snd} \ (\_, y) & \quad = y \\
id & \quad :: \ a \to a \\
id \ x & \quad = x \\
(\cdot ) & \quad :: \ (b \to c) \to (a \to b) \to (a \to c) \\
(f \ . \ g) \ x & \quad = f \ (g \ x) \\
\text{head} \ , \ \text{last} & \quad :: \ [a] \to a \\
\text{head} \ (x:\) & \quad = x \\
\text{last} \ [x] & \quad = x \\
\text{last} \ (\_ : x) & \quad = \text{last} \ x \\
\text{tail} \ , \ \text{init} & \quad :: \ [a] \to [a] \\
\text{tail} \ (\_ : x) & \quad = x \\
\text{init} \ [x] & \quad = [\] \\
\text{init} \ (x : x) & \quad = x : \text{init} \ x \\
\text{null} & \quad :: \ [a] \to \text{Bool} \\
\text{null} \ [\] & \quad = \text{True} \\
\text{null} \ (\_ : \_ ) & \quad = \text{False} \\
(\cdot + \cdot) & \quad :: \ [a] \to [a] \to [a] \\
[\] \ (\_ : x : x) & \quad = x : (x + y) \\
(x : x) \ (\_ : y : y) & \quad = x : (x + y) \\
\text{map} & \quad :: \ (a \to b) \to [a] \to [b] \\
\text{map} \ f \ [\] & \quad = [\] \\
\text{map} \ f \ (x : x) & \quad = f \ x : \text{map} \ f \ x \\
\text{filter} & \quad :: \ (a \to \text{Bool}) \to [a] \to [a] \\
\text{filter} \ [\] & \quad = [\] \\
\text{filter} \ p \ (x : x) & \quad = x : \text{filter} \ p \ x \\
| \ & \quad \text{otherwise} = \text{filter} \ p \ x \\
\text{concat} & \quad :: \ [[a]] \to [a] \\
\text{concat} & \quad = \text{foldr} \ (\cdot + \cdot) \ [\] \\
\text{length} & \quad :: \ [a] \to \text{Int} \\
\text{length} & \quad = \text{foldl} \ (\cdot x \to x + 1) \ 0 \\
(\cdot \cdot) & \quad :: \ [a] \to \text{Int} \to a \\
(x : \_ ) \ (\cdot \cdot) \ 0 & \quad = x \\
(x : x) \ (\cdot \cdot) \ n & \quad | \ n \geq 0 = x \cdot \cdot \ (n-1) \\
(\_ : \_ ) \ (\cdot \cdot) \ n & \quad \text{otherwise} = \text{error} \ "\text{Prelude} :: \text{negative\ index}" 
\end{align*}
\]
[]    ! ! _ = error "Prelude.!!: index too large"

foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)

iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)

take    :: Int -> [a] -> [a]
take n _ | n <= 0 = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs

drop    :: Int -> [a] -> [a]
drop n xs | n <= 0 = xs
drop _ [] = []
drop n (_:xs) = drop (n-1) xs

zip    :: [a] -> [b] -> [(a,b)]
zip = zipWith (\a b -> (a,b))

zipWith :: (a->b->c) -> [a]->[b]->[c]
zipWith z (a:as) (b:bs) = z a b : zipWith z as bs
zipWith _ _ _ = []

takeWhile :: (a->Bool) -> [a] -> [a]
takeWhile p [] = []
takeWhile p (x:xs)
  | p x = x : takeWhile p xs
  | otherwise = []

dropWhile :: (a->Bool) -> [a] -> [a]
dropWhile p [] = []
dropWhile p (x:xs)
  | p x = dropWhile p xs
  | otherwise = x:xs

flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x

until :: (a -> Bool) -> (a -> a) -> a -> a
until p f x = if p x then x else until p f (f x)

sort :: Ord a => [a] -> [a]
sort xs = ...

and, or :: [Bool] -> Bool
and = ...
or = ...

ord :: Char -> Int
chr :: Int -> Char
toUpper, toLower :: Char -> Char
isAscii, isDigit :: Char -> Bool
isUpper, isLower :: Char -> Bool
Useful Prolog Predicates

append([], L, L)
append([X|L1], L2, [X|L3]) :-
    append(L1, L2, L3).

member(X, [X|_]).
member(X, [_|Y]) :- member(X, Y).

delete_one(X,[X|Z],Z).
delete_one(X,[V|Z],[V|Y]) :-
    X \= V, delete_one(X,Z,Y).

permutation(X,[Z|V]) :-
    delete_one(Z,X,Y),
    permutation(Y,V).
prefixation([],[]).

• setof(X,Goal,List)
    – List is a collection of Xs for which Goal is true.
    – List is sorted and contains no duplicates.
• bagof(X,Goal,List)
    – List may contain duplicates.
• setof and bagof will fail if no Goals succeed.
• findall(X,Goal,List)
    – findall will return [] if no Goals succeed.
• name(Atom,List)
    – name converts between an atom and its string (list of ASCII values) representation.
• asserta(G) and assertz(G)
    – Adds G to the database, first or last, respectively.
Useful Ruby Methods

- The module Comparable:

```ruby
module Comparable
  def ==(other)
  def >(other)
  def >=(other)
  def <=(other)
  def <(other)
end
```

- The module Enumerable:

```ruby
module Enumerable
  def each_with_index { |obj, index| ... }

  def collect() { |x| ... }

  def sort()

  def member?(obj)

  def inject(init) { |total, obj| ... }

  def find { |obj| ... }
end
```

- Regular expressions:

  - `str.scan(RE)` iterates through the string `str` matching the regular expression. It can be called as a function, returning an array of results. It can also be called with a block attached in which case the block gets invoked for every match with the matched substring as the argument.
  - `str.match(RE)` returns the first substring of `str` that matches, or `nil` if there’s no match.
  - `str.split(RE)` splits `str` wherever the regular expression matches. The results are returned as an array.
  - `str.sub(RE, rep)` returns a copy of `str` where the first occurrence of `RE` has been replaced with `rep`. `str.sub(RE) { |x| ... }` instead passes a block to the method, and the block returns what should be replaced. `gsub` is similar but replaces all matches in the string. `sub!` and `gsub!` perform the substitutions in-place.
- There are some standard abbreviations:
  * \d ≡ [0-9]
  * \D ≡ [\^0-9]
  * \s ≡ [\t\r\n\f]
  * \S ≡ [\^\t\r\n\f]
  * \w ≡ [A-Za-z0-9]
  * \W ≡ [\^A-Za-z0-9]

- The module Integer:
  - int.upto(limit) {i| block} => int
    Iterates block, passing in integer values from int up to and including limit.
  - int.downto(limit) {i| block} => int
    Iterates block, passing decreasing values from int down to and including limit.
  - int.step(limit, step) {i | block}
    Invokes block with the sequence of numbers starting at int, incremented by step on each call. The loop finishes when the value to be passed to the block is greater than limit (if step is positive) or less than limit (if step is negative).
  - int.times {i| block}
    Iterates block int times, passing in values from zero to int-1.