# Useful Functions from the Haskell Standard Prelude

```
fst,snd          :: (a,b) -> a
fst (x,_)        = x
snd (_,y)        = y

id               :: a -> a
id    x          = x

(.)              :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x        = f (g x)

head,last        :: [a] -> a
head (x:_)         = x

last [x]           = x
last (_:xs)        = last xs

tail,init        :: [a] -> [a]
tail (_:xs)        = xs

init [x]           = []
init (x:xs)        = x : init xs

null             :: [a] -> Bool
null []          = True
null (_:_)       = False

(++)             :: [a] -> [a] -> [a]
[]     ++ ys     = ys
(x:xs) ++ ys     = x : (xs ++ ys)

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

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

concat           :: [[a]] -> [a]
concat           = foldr (++) []

length           :: [a] -> Int
length           = foldl (\x _ ->x+1) 0

(!!)             :: [a] -> Int -> a
(x:_)  !! 0      = x
(_:xs) !! n | n>0 = xs !! (n-1)
(_:_)  !! _      = error "Prelude.!!: negative index"
```

```
[]     !! _       = 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).
permutation([],[]).
```

- `setof(X,Goal,List)`
  - `List` is a collection of `X`s 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 `Goal`s succeed.
- `findall(X,Goal,List)`
  - `findall` will return `[]` if no `Goal`s 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`:

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

- The module `Enumerable`:

```
module Enumerable
  // Calls block with two arguments, the item and its index, for each item.
  def each_with_index {|obj,index| ...}

  // Returns a new array containing the results of running the supplied block
  // for every element.
  def collect() {|x| ...}

  // Returns an array containing the elements sorted according to their <=> method.
  def sort()

  // Returns true if any element equals "obj".
  def member?(obj)

  // Combines the elements of the collection by applying the supplied
  // block to an accumulator value "total" and each element in turn.
  def inject(init) {|total,obj| ...}

  // Passes each entry of the collection to the block and returns the
  // first for which block doesn't return "false". Returns nil if no
  // object matched.
  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 occurence 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.