Chapter 5 - List Comprehensions
Set Comprehensions

In mathematics, the comprehension notation can be used to construct new sets from old sets.

\[ \{ x^2 \mid x \in \{1\ldots5\} \} \]

The set \( \{1,4,9,16,25\} \) of all numbers \( x^2 \) such that \( x \) is an element of the set \( \{1\ldots5\} \).
Lists Comprehensions

In Haskell, a similar comprehension notation can be used to construct new lists from old lists.

\[ \{ x^2 \mid x \leftarrow [1..5] \} \]

The list \([1,4,9,16,25]\) of all numbers \(x^2\) such that \(x\) is an element of the list \([1..5]\).
Note:

- The expression $x \leftarrow [1..5]$ is called a **generator**, as it states how to generate values for $x$.

- Comprehensions can have **multiple** generators, separated by commas. For example:

```
> [(x,y) | x ← [1,2,3], y ← [4,5]]
[(1,4), (1,5), (2,4), (2,5), (3,4), (3,5)]
```
Changing the order of the generators changes the order of the elements in the final list:

\[
> [(x, y) | y \leftarrow [4, 5], x \leftarrow [1, 2, 3]]
\]

\[
[(1, 4), (2, 4), (3, 4), (1, 5), (2, 5), (3, 5)]
\]

Multiple generators are like nested loops, with later generators as more deeply nested loops whose variables change value more frequently.
For example:

> [(x,y) | y ← [4,5], x ← [1,2,3]]

[(1,4), (2,4), (3,4), (1,5), (2,5), (3,5)]

x ← [1,2,3] is the last generator, so the value of the x component of each pair changes most frequently.
Dependant Generators

Later generators can depend on the variables that are introduced by earlier generators.

\[((x,y) \mid x \leftarrow [1..3], y \leftarrow [x..3])\]

The list \[((1,1),(1,2),(1,3),(2,2),(2,3),(3,3))\]
of all pairs of numbers \((x,y)\) such that \(x,y\) are elements of the list \([1..3]\) and \(y \geq x\).
Using a dependant generator we can define the library function that **concatenates** a list of lists:

\[
\text{concat} :: [[a]] \rightarrow [a] \\
\text{concat } xss = [x | xs \leftarrow xss, x \leftarrow xs]
\]

For example:

\[
> \text{concat } [[1,2,3],[4,5],[6]] \\
[1,2,3,4,5,6]
\]
Guards

List comprehensions can use *guards* to restrict the values produced by earlier generators.

\[ [x \mid x \leftarrow [1..10], \text{even } x] \]

The list \([2,4,6,8,10]\) of all numbers \(x\) such that \(x\) is an element of the list \([1..10]\) and \(x\) is even.
Using a guard we can define a function that maps a positive integer to its list of factors:

```haskell
factors :: Int → [Int]
factors n = [x | x ← [1..n], n `mod` x == 0]
```

For example:

> factors 15

[1,3,5,15]
A positive integer is prime if its only factors are 1 and itself. Hence, using factors we can define a function that decides if a number is prime:

\[
\text{prime} :: \text{Int} \rightarrow \text{Bool} \\
\text{prime } n = \text{factors } n == [1,n]
\]

For example:

\[
> \text{prime 15} \\
\text{False} \\
> \text{prime 7} \\
\text{True}
\]
Using a guard we can now define a function that returns the list of all primes up to a given limit:

```haskell
primes :: Int → [Int]
primes n = [x | x ← [2..n], prime x]
```

For example:

```haskell
> primes 40
[2,3,5,7,11,13,17,19,23,29,31,37]
```
The Zip Function

A useful library function is `zip`, which maps two lists to a list of pairs of their corresponding elements.

\[
\text{zip} :: [a] \rightarrow [b] \rightarrow [(a,b)]
\]

For example:

\[
> \text{zip} \ [\text{'a'},\text{'b'},\text{'c'}] \ [1,2,3,4] \\
[(\text{'a'},1), (\text{'b'},2), (\text{'c'},3)]
\]
Using zip we can define a function returns the list of all pairs of adjacent elements from a list:

```
pairs :: [a] → [(a,a)]
pairs xs = zip xs (tail xs)
```

For example:

```
> pairs [1,2,3,4]
[(1,2),(2,3),(3,4)]
```
Using pairs we can define a function that decides if the elements in a list are sorted:

```haskell
sorted :: Ord a ⇒ [a] → Bool
sorted xs = and [x ≤ y | (x,y) ← pairs xs]
```

For example:

```haskell
> sorted [1,2,3,4]
True
> sorted [1,3,2,4]
False
```
Using zip we can define a function that returns the list of all *positions* of a value in a list:

```haskell
positions :: Eq a ⇒ a → [a] → [Int]
positions x xs =
    [i | (x',i) ← zip xs [0..n], x == x']
where n = length xs - 1
```

For example:

```
> positions 0 [1,0,0,1,0,1,1,0]
[1,2,4,7]
```
String Comprehensions

A **string** is a sequence of characters enclosed in double quotes. Internally, however, strings are represented as lists of characters.

"abc" :: String

Means ['a','b','c'] :: [Char].
Because strings are just special kinds of lists, any polymorphic function that operates on lists can also be applied to strings. For example:

```haskell
> length "abcde"
5

> take 3 "abcde"
"abc"

> zip "abc" [1,2,3,4]
[('a',1),('b',2),('c',3)]
```
Similarly, list comprehensions can also be used to define functions on strings, such as a function that counts the lower-case letters in a string:

```haskell
lowers :: String → Int
lowers xs =
  length [x | x ← xs, isLower x]
```

For example:

```haskell
> lowers "Haskell"
6
```
A triple \((x, y, z)\) of positive integers is called **pythagorean** if \(x^2 + y^2 = z^2\). Using a list comprehension, define a function

\[
\text{pyths} :: \text{Int} \rightarrow [(\text{Int},\text{Int},\text{Int})]
\]

that maps an integer \(n\) to all such triples with components in \([1..n]\). For example:

\[
> \text{pyths 5} \\
\quad [(3,4,5),(4,3,5)]
\]
A positive integer is **perfect** if it equals the sum of all of its factors, excluding the number itself. Using a list comprehension, define a function

```haskell
perfects :: Int → [Int]
```

that returns the list of all perfect numbers up to a given limit. For example:

```haskell
> perfects 500
[6,28,496]
```
(3) The **scalar product** of two lists of integers $xs$ and $ys$ of length $n$ is given by the sum of the products of the corresponding integers:

$$
\sum_{i=0}^{n-1} (xs_i \times ys_i)
$$

Using a list comprehension, define a function that returns the scalar product of two lists.