FACULTY OF COMPUTER SCIENCE

UNIVERSITAS INDONESIA

Veritas, Probitas, Iustitia

Functional Programming
Semester Gasal 2019 Term 1

Week 5: List Comprehension and Lazy Evaluation

(Ditulis ulang dari HaskellWiki)

List comprehensions are syntactic sugar like the expression

```
import Data.Char (toUpper)
[toUpper c | c <- s]
```

where s :: String is a string such as "Hello". Strings in Haskell are lists of characters; the generator c <- s feeds each character of s in turn to the left-hand expression toUpper c, building a new list. The result of this list comprehension is "HELLO". (Of course, in this simple example you would just write map toUpper s.)

One may have multiple generators, separated by commas, such as

```
[(i,j) | i <- [1,2], j <- [1..4] ]
```

yielding the result: `[(1,1),(1,2),(1,3),(1,4),(2,1),(2,2),(2,3),(2,4)]`

Note how each successive generator refines the results of the previous generator. Thus, if the second list is infinite, one will never reach the second element of the first list. For example,

```
take 10 [ (i,j) | i <- [1,2], j <- [1..] ]
```

Yields: `[(1,1),(1,2),(1,3),(1,4),(1,5),(1,6),(1,7),(1,8),(1,9),(1,10)]`

In such a situation, a nested sequence of list comprehensions may be appropriate. For example,

```
take 3 [ [ (i,j) | i <- [1,2] ] | j <- [1..] ]
```

Yields: `[[(1,1),(2,1)], [(1,2),(2,2)], [(1,3),(2,3)]]`

One can also provide boolean guards. For example,

```
take 10 [ (i,j) | i <- [1..],  j <- [1..i-1], gcd i j == 1 ]
```

Yields: `[(2,1),(3,1),(3,2),(4,1),(4,3),(5,1),(5,2),(5,3),(5,4),(6,1)]`

Finally, one can also make local let declarations. For example,

```
take 10 [ (i,j) | i <- [1..], let k = i*i, j <- [1..k] ]
```

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

Lazy evaluation is a method to evaluate a Haskell program. It means that expressions are not evaluated when they are bound to variables, but their evaluation is **deferred** until their results are needed by other computations. In consequence, arguments are not evaluated before they are passed to a function, but only when their values are actually used. Technically, lazy evaluation means call-by-name plus Sharing. A kind of opposite is eager evaluation.

While lazy evaluation has many advantages, its main drawback is that memory usage becomes hard to predict. The thing is that while two expressions, like `2+2 :: Int` and `4 :: Int`, may denote the same value 4, they may have very different sizes and hence use different amounts of memory.

An extreme example would be the infinite list `1 : 1 : 1 …` and the expression `let x = 1:x in x`. The latter is represented as a cyclic graph, and takes only finite memory, but its denotation is the former infinite list.

Latihan:

1. Uraikan langkah evaluasi dari ekspresi berikut: **[x+y | x <- [1 .. 4], y <- [2 .. 4], x > y]**
2. Buatlah fungsi divisor yang menerima sebuah bilangan bulat ***n*** dan mengembalikan list bilangan bulat positif yang membagi habis ***n***, Contoh:

   ```
   LatihanLazy> divisor 12
   [1,2,3,4,6,12]
   ```
3. Buatlah definisi *quick sort* menggunakan list comprehension.
4. Buatlah definisi infinite list untuk permutation.
5. Buatlah definisi untuk memberikan infinite list dari bilangan prima menerapkan algoritma *Sieve of Erastothenes*.
6. Buatlah definisi infinite list dari *triple pythagoras*. List tersebut terdiri dari element triple bilangan bulat positif yang mengikut persamaan pythagoras $\mathbf{x^2 + y^2 = z^2}$. Contoh:

   ```
   LatihanLazy > pythaTriple
   [(3,4,5),(6,8,10),(5,12,13),(9,12,15),(8,15,17),(12,16,20) … ]
   ```

   Perhatian urutan penyusun list comprehension nya, coba mulai dari variable z!