Functional programming in R (R'19)

TD 3: Control and Data flow

Exercise 1. What does the following expressions evaluate to, why/how, and what are their side effects (such as variable assignments)

1. `{y ← {x ← 2; y ← x-1}; y ← x+y}`
2. `{z ← y ← x ← 1; {y ← z + 1} + 3} + 2`
3. `{x ← 1; for (i in 1:3) {x ← x+i}}`
4. `{i ← 1; repeat {for (j in 1:2) i ← (i+1)+j; if (i %% 4 == 0) break};i}`

Exercise 2. Define a (higher-order) function `cycle` : $\alpha^* \to$ `int` $\to \alpha$ which for a given vector `v` returns a function `V` ← `cycle(v)` that allows to access the elements of `v` in a manner analogous to how recycling works. For instance, if `v` ← `c(2,5,8)` and `V` ← `cycle(v)`, then `V(1)` ↦ 2, `V(2)` ↦ 5, `V(3)` ↦ 8, `V(4)` ↦ 2, `V(5)` ↦ 5, `V(6)` ↦ 8, etc.

Exercise 3. Implement the function `rev` using loops (see the help page of `rrev` for detail on the function).

Exercise 4. Use the function `cat` to write a function `xmas` that takes an integer n and prints out a X-mas tree of height n. For instance, a simple x-mas tree of height 4 is (try placing ornaments `o` and `!` at strategic places)

```
   *
  ***
 *****
*******
```

Exercise 5. Implement the function `strsplit` : `chr` × `chr` → `chr`* that takes a string `s` of characters and a separator `p` (a single character) and returns a list of maximal fragments of the string `s` that does not contain `p`. For instance, `strsplit('abc|serr|ds','|')` ↦`'abc' 'serr' 'ds'`. Use only functions, `nchar` and `substr`.

Exercise 6. Write a function `flatten` : `chr`* × `chr`* → `log`* that takes two vectors of strings `v` and `w` and returns a Boolean vector `f` of length equal to the length of `v` such that `f`[`i`] is `TRUE` if and only if the string `v[i]` is present in the vector `w`. For instance, `flatten(c('a','b','c','d'),c('a','c','e'))`↦ `T F T F`.

Exercise 7. Propose vectorized version of the following procedures (without `ifelse` or any `apply` function)

1.
```
function (v,w) {
    if (length(v) != length(w))
       FALSE
    else
       for (i in 1:length(v)) {
          if (v[i] != w[i]) {
             return(FALSE)
          }
       }
       TRUE
}
```

2.
```
function (w,v) {
   V ← cycle(v)
   W ← cycle(w)
   n ← max(length(v),length(u))
   u ← vector()
   for (i in 1:n) {
      u[i] ← V(i) * W(i) + 2
   }
   u
}
```

3.
```
function (v) {
   u ← vector()
   for (i in 1:length(v)) {
      if (i %% 2 == 1){
         u[i] ← v[i]
      } else
         u[i] ← -v[i]
      }
   u
}
```

4.
```
function (v,w,u) {
   n ← max(length(v),length(w),length(u)
   x ← vector()
   V ← cycle(v)
   W ← cycle(w)
   U ← cycle(u)
   for (i in 1:n) {
      if (i %% 3 == 1) {
         x[i] ← V(i) + 2*W(i)
      else if (i %% 3 == 2) {
         x[i] ← W(i) - U(i)
      else {
         x[i] ← 3*U(i) + V(i) - 2*W(i)
      }
   }
   x
}
```

5.
```
function (v,w) {
   n ← max(length(v),length(w))
   u ← vector()
   V ← cycle(v)
   W ← cycle(w)
   for (i in 1:n) {
      u[2*i-1] ← V(i)
      u[2*i] ← W(i)
   }
   u
}
```

6*.
```
function (v) function (w) {
    if (length(v) >= length(w)) {
       v
    } else {
       n ← length(w)
       u ← vector()
       for (i in 1:n) {
          u[i] ← v[(i-1)%%n+1]
       }
       u
    }
}
```

Exercise 8. Use functions from the `apply` family to propose vectorised version of the following functions

1.
```
function (v,f,g) {
    u ← vector()
    for (i in 1:length(v)) {
       u[i] ← g(f(v[i]))
    }
    u
}
```

2.
```
function (v,f,g) {
    l ← list()
    for (i in 1:length(v)) {
       u ← vector(length(v))
       for (j in 1:length(v)) {
          u[j] ← g(i) * f(v[j])
       }
       l[[i]] ← u
    }
    l
}
```

Exercise 9. Propose conversion functions between list and factors, vectors (matrices), and data frames.

Exercise 10. [*] Propose a (vectorized) implementation of the built-in function `ifelse`. In particular,

```
x ← c(1,2,3,4,6,7,8,9); ifelse(x %% 2 == 0, c("E","e"), c("O","o","."))
```

should return the vector `"O" "e" "." "e" "E" "." "E" "o"`.