Flow controls (1)

- **Conditional execution**
  - `if`
  - `ifelse`
  - `Switch`
if-statement

> grade = "PASS" # default value for grade
> score = 50 # actual score
> if (score<60) grade = "FAIL"
> grade # actual grade
[1] "FAIL"

> score<-50
> if (score<60) grade = "FAIL" else grade = "PASS"
> grade
[1] "FAIL"
ifelse-function

> score <- 50
> grade <- ifelse(score < 60, "FAIL", "PASS")
> grade
[1] "FAIL"
if-else-statement

```r
cpy score <- 50
> if (score < 60) grade = "FAIL" else grade = "PASS"
> grade
[1] "FAIL"
```
switch-function

```r
> lett <- c("b", "QQ", "a", "A", "bb")
> for(ch in lett)
+ cat(ch, ":", switch(EXPR = ch, a=1, A=1, b=2, B=2, "No match!"), "\n")
  b : 2
  QQ : No match!
  a : 1
  A : 1
  bb : No match!
```
Flow controls (2)

- Repetitive execution
  - for
  - repeat
  - while
for-loop

```r
> for(i in 1:5) print(1:i)
[1] 1
[1] 1 2
[1] 1 2 3
[1] 1 2 3 4
[1] 1 2 3 4 5
```

```r
> for(i in c(1,2,3,4,5)) print(1:i)
[1] 1
[1] 1 2
[1] 1 2 3
[1] 1 2 3 4
[1] 1 2 3 4 5
```
repeat and while commands

- **repeat expr**
  - A *repeat* loop repeated executes the expression until explicitly terminated.

- **while (cond) expr**
  - A *while* loop continues execution of the expression while the condition (cond) holds true.
Vectorization

- In R, loops are very inefficiently implemented. Instead, vectorization is used wherever applicable.

```r
> score <- c(80, 45, 55, 90, 75)
> n <- length(score)
> grade <- integer(n)
> for (i in 1:n) {
+  grade[i] <- ifelse(score[i] < 60, 0, 1)
} > grade
[1] 1 0 0 1 1
```
Write your own functions

- A function is defined by using the keyword `function`, followed by an opening parenthesis, a list of formal arguments (separated by commas), and a closing parenthesis, and then by the expressions that for the body of the functions.

- The value returned by a function is either the value that is explicitly returned by a call “return” or it is simply the value of the last expression.

```r
> sq1 <- function(x) x * x
> sq1(5)
[1] 25

> sq2 <- function(x) 
  return(x * x)
> sq2(5)
[1] 25

> sq3 <- function(x) {
  y <- x * x
  return(y)
}
> sq3(5)
[1] 25
```
A function for normal likelihood

- In a forward reasoning, given parameter $B$, we use the conditional probability $P(A \mid B)$ to reason about outcome $A$. In a backward reasoning, however, outcome $A$ is given and we use the likelihood function $L(B \mid A)$ to reason about parameter $B$.

- Formally, a likelihood function is a conditional probability function considered as a function of its second argument, with its first argument held fixed.

- Normal likelihood (is or proportional to)

$$L = \left(2\pi\sigma^2\right)^{-n/2} \exp \left( - \frac{\sum_{i=1}^{n} (y_i - \mu)^2}{2\sigma^2} \right)$$
A function of the logarithmic normal likelihood

\[
\log L = \left(-\frac{n}{2}\right) \log \left(2\pi\sigma^2\right) + \left(-\frac{1}{2\sigma^2}\right) \sum_{i=1}^{n} (y_i - \mu)^2
\]

```r
loglike <- function(mu, sigma, yobs) {
  n <- length(yobs)
  var <- sigma * sigma
  logL <- 0.5*n*log(2.0*pi*var) + sum((yobs-mu)^2)/(2.0*var)
  return(-logL)
}
```
Calculate the likelihood

- Generate a sample from a normal distribution

```r
> mu <- 1.0
> sigma <- 1.2
> y <- rnorm(n = 100, + mean = mu, sd = sigma)
> mean(y)
[1] 0.9129361
> var(y)
[1] 1.506054
```

- Calculate the likelihood

```r
> mu <- 1.0
> sigma <- sqrt(1.2)
> logL <- loglike(mu, sigma, y)
> logL
[1] -163.4505
> exp(logL)
[1] 1.033612e-71
```
Functions with default values

- Logarithmic normal likelihood with a fixed variance
  > loglike <- function(mu, sigma=1, yobs) { …… }

- calculates the likelihoods for a grid of values for mu with a fixed variance.
  > likemu <- function(vmu, yobs) {
  >   m <- length(vmu)
  >   like <- numeric(m)
  >   for (i in 1:m) {
  >     like[i] <- exp(loglike(mu = vmu[i], yobs = yobs))
  >   }
  >   return(like)
  > }
Likelihoods for a grid of mu values

> vmu<-seq(-2,2,0.1)
> lmu<-likemu(vmu=vmu,yobs=y)
> plot(vmu,lmu,type="h")
Functions for Binary operators

- A binary operation takes two values, such as addition (+), subtraction (-), multiplication (*), and division (/).

- In R, binary operators also include matrix multiplication `%*%` and outer product `%o%`.

- A binary operator is a function, e.g.,
  ```r
  > get("+")
  function (e1, e2) .Primitive("+")
  ```

- We write, for example, `1 + 2`, rather than `+(1, 2)`. 
Define a binary `%m%`

- Binary operator: \( a \%m\% b = ab-b \).

- R code:
  
  ```r
  > "\%m\%" <- function(a,b) a*b-b
  
  > 1 %m% 2
  [1] 0
  > 5 %m% 6
  [1] 24
  ```
Recursive functions (1)

- A recursive function is a function that calls itself.
- Consider computing a factorial: \( n! = n \times (n-1)! \)
  - A recursive function can be used here, because, to compute \( n! \), one can compute \((n - 1)!\), and then multiplied by \( n \).

Algorithm:
- \( 0! = 1 \)
- \( 1! = 1 \times 0! = 1 \times 1 = 1 \)
- \( 2! = 2 \times 1! = 1 \times 1 = 2 \)
- \( 3! = 3 \times 2! = 3 \times 2 = 6 \)
- \( 4! = 4 \times 3! = 4 \times 6 = 24 \)
Recursive functions (2)

- R code:

```r
factorial <- function (n) {
    if (n == 0) return(1)
    else return(n * factorial(n - 1))
}
```

- > source("factorial.R")
- > factorial(0)
  [1] 1
- > factorial(1)
  [1] 1
- > factorial(2)
  [1] 2
- > factorial(3)
  [1] 6
- > factorial(4)
  [1] 24
- > factorial(10)
  [1] 3628800
Some issues related to R programming*

- Lexical scope
- Exception handling
- Classes and generic functions

* Read course note Chapter 3 (section 3.3)