4장. R 프로그래밍

if~else 문장

if 구문은 조건식을 이용해 단독으로 사용될 수 있습니다. 경우에 따라 else if 블록을 이용해 추가 조건식을 정할 수 있으며, if 블록 또는 else if 블록에서 만족하는 조건식을 찾지 못할 때 실행되는 else 블록을 가질 수 있습니다.

```
if(조건식_1) {
    # 조건식_1이 참일 경우
} else if(조건식_2) {
    # 조건식_2가 참일 경우 실행, else if 블록 생략 가능
} else {
    # 모든 조건식이 거짓일 경우 실행, else 블록 생략 가능
}
```

if~else 문장


```
> number <- 10
> if( (number %% 2) == 0) {
+   print("짝수입니다")
+ }
[1] "짝수입니다"
> print("if 블록의 바깥입니다.")
[1] "if 블록의 바깥입니다."
```

```
> number <- 9
> if( (number %% 2) == 0) {
+   print("짝수입니다")
+ }
> print("if 블록의 바깥입니다.")
[1] "if 블록의 바깥입니다."
```

if~else 문장


```
> number <- 10
> if( (number %% 2) == 0) {
+   print("짝수입니다.")
+ }else {
+   print("홀수입니다.")
+ }
[1] "짝수입니다."
```

```
> number <- 9
> if( (number %% 2) == 0) {
+   print("짝수입니다.")
+ }else {
+   print("홀수입니다.")
+ }
[1] "홀수입니다."
```

if~else 문장


```
> jumsu <- 80
> if(jumsu >=90) {
+   print("A")
+ } else if(jumsu >= 80) {
+   print("B")
+ } else if(jumsu >= 70) {
+   print("C")
+ } else if(jumsu >= 60) {
+   print("D")
+ } else {
+   print("Fail")
+ }
[1] "B"
```

ifelse() 함수

ifelse() 함수는 test 요소가 TRUE 또는 FALSE인지 여부에 따라 yes 또는 no 중에서 선택된 요소로 채워진 test와 동일한 모양의 값을 반환합니다. ifelse() 함수는 간단한 if~else 블록을 작성할 때 사용합니다.

```
ifelse(test, yes, no)
```

```
> number <- 3
> ifelse((number%%2)==0, "짝수", "홀수")
[1] "홀수"

> number <- 2
> ifelse((number%%2)==0, "짝수", "홀수")
[1] "짝수"
```

구문에서...

- test : TRUE 또는 FALSE로 바뀔 수 있는 객체입니다.
- yes : test의 값이 TRUE일 경우 리턴되는 값입니다.
- no : test의 값이 FALSE일 경우 리턴되는 값입니다.

ifelse() 함수

ifelse() 함수는 벡터를 이용하여 한 번에 여러 개 데이터의 조건식을 처리할 수 있습니다.

```
> a <- c(5,7,2,9)
> ifelse(a %% 2 == 0,"even","odd")
[1] "odd"  "odd"  "even" "odd"
```

if~else 구문은 조건식에 비교하는 객체의 길이가 1보다 클 수 없습니다.

```
> number <- c(5,7,2,9)
> if( (number %% 2) == 0) {
+   print("짝수입니다.")
+ }else {
+   print("홀수입니다.")
+ }
[1] "홀수입니다."
Warning message:
In if ((number%%2) == 0) { :
  the condition has length > 1 and only the first element will be used
```

switch() 함수

다른 언어의 switch 문처럼 R도 switch() 함수 형태로 비슷한 구문을 사용합니다.

```
switch (statement, list)
```

여기에서는 statement 명령문이 평가되고 이 값에 따라 리스트(list)의 해당 항목이 반환됩니다.

```
> switch(2,"red","green","blue")
[1] "green"

> switch(1,"red","green","blue")
[1] "red"
```

switch() 함수

숫자 값이 범위를 벗어났거나(리스트의 항목 수보다 크거나 1보다 작 으면) NULL이 반환됩니다.

```
> x <- switch(4,"red","green","blue")
> x
NULL

> x <- switch(0,"red","green","blue")
> x
NULL
```

명령문의 결과는 문자열 일 수도 있습니다. 이 경우 이름이 지정된 항목의 값과 일치하는 항목이 반환됩니다.

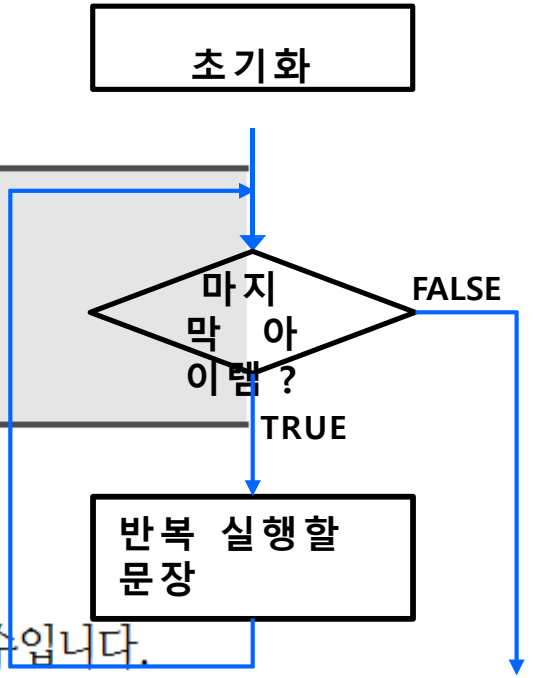
```
> switch("color", "color"="red", "shape"="square", "length"=5)
[1] "red"

> switch("length", "color"="red", "shape"="square", "length"=5)
[1] 5
```

반복문 - for

for 문장은 벡터의 데이터를 모두 소비할 때 까지 실행하는 반복문입니다. sequence 벡터의 내용을 val 변수에 저장하고 statement를 실행시킵니다. 만일 더 이상 sequence 벡터에 남아있는 데이터가 없을 경우 반복문은 종료합니다.

```
for (val in sequence) {
    statement
}
```



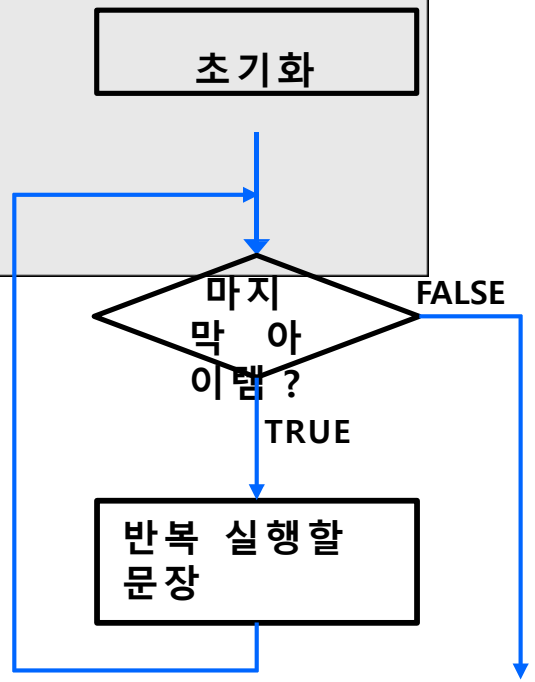
구문에서...

- val : 반복문이 실행되는 동안 sequence의 값들 중 하나를 저장할 변수입니다.
- sequence : 벡터 데이터입니다.

반복문 – for

다음 코드는 벡터 x에서 짝수의 개수를 세는 for문의 예입니다.

```
> x <- c(2,5,3,9,8,11,6)
> count <- 0
> for (val in x) {
+   if(val %% 2 == 0)  count = count+1
+ }
> print(count)
[1] 3
```



반복문 – for

factorial.R

```
num = as.integer(readline(prompt="Enter a number: "))
factorial = 1
if(num < 0) {
    print("Sorry, factorial does not exist for negative numbers")
} else if(num == 0) {
    print("The factorial of 0 is 1")
} else {
    for(i in 1:num) {
        factorial = factorial * i
    }
    print(paste("The factorial of", num ,"is",factorial))
}
```

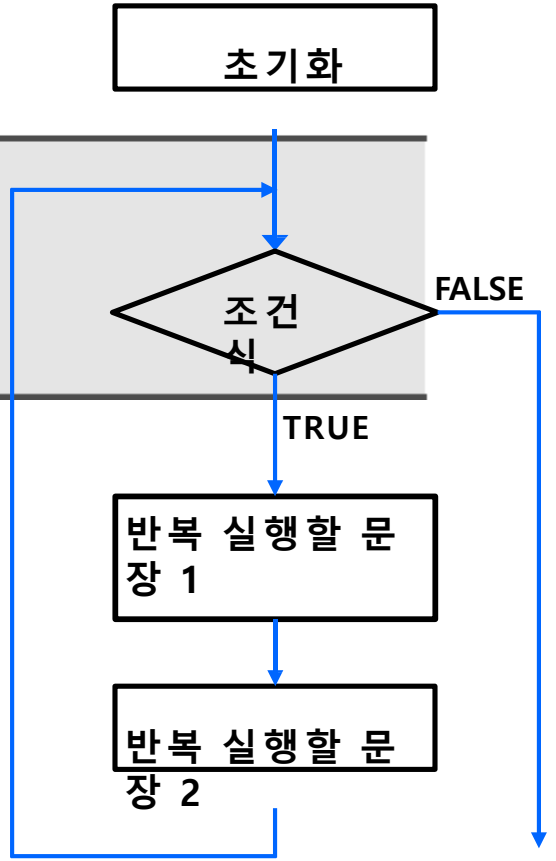
위의 R 스크립트를 R Console에서 실행시키면 구하고 싶은 팩토리얼만 입력하면 결과를 얻을 수 있습니다.

```
> source("factorial.R")
Enter a number: 9
[1] "The factorial of 9 is 362880"
```

반복문 – while

while 문장은 조건식이 참일 동안 statement가 실행됩니다. 조건식의 결과는 TRUE 또는 FALSE 여야 합니다.

```
while (조건식) {
    statement
}
```



반복문 - while

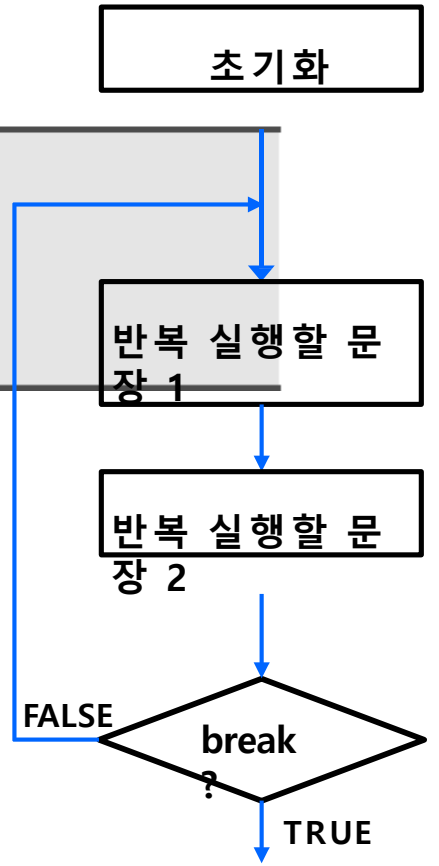
다음 코드는 i값을 i값이 6보다 작을 때까지 i를 출력하며 하나씩 증가시킵니다. while문 안에서 i값은 변경시켜 while 문의 조건식이 언젠간 종료되도록 해야 합니다.

```
> i <- 1
> while (i < 6) {
+   print(i)
+   i = i+1
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

반복문 - repeat

repeat 루프는 코드 블록을 여러 번 반복하는 데 사용됩니다. repeat는 루프를 종료시키는 조건 문장이 없습니다. 그러므로 반복문 내부에 종료 조건을 명시하고 break 문을 사용해 반복을 종료해야 합니다. 그렇지 않으면 무한 루프가 됩니다.

```
repeat {
   statement
}
```



반복문 - repeat

다음 코드는 repeat 문장의 예입니다. x값이 6일 경우 break 문을 실행시켜 반복문을 종료시키도록 합니다.

```
> x <- 1
> repeat {
+   print(x)
+   x = x+1
+   if (x >= 6){
+       break
+   }
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

탈출문 - break

break는 반복문을 완전히 빠져 나옵니다. 다음 구문에서 i가 5일 경우 반복문은 더 이상 실행되지 않습니다.

```
> for(i in 1:10) {
+   if(i==5)
+     break
+   print(i)
+ }
[1] 1
[1] 2
[1] 3
[1] 4
```

탈출문 - next

next는 반복문에서 아무것도 하지 않고 다음 반복으로 건너뜁니다. 다음 구문에서 i가 5일 경우 아무것도 하지 않고 다음 반복을 진행합니다.

```
> for(i in 1:10) {
+   if(i==5)
+     next
+   print(i)
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

연산자

산술 연산자

연산자	설명
+	덧셈
-	뺄셈
*	곱셈
/	나눗셈
^	지수
%%	나눗셈 후 나머지
%/%	몫(정수)

관계 연산자

연산자	설명
<	작다
>	크다
<=	작거나 같다
>=	크거나 같다
==	같다
!=	다르다

논리 연산자

연산자	설명
!	논리 NOT
&	엘리먼트 단위 논리 AND
&&	논리 AND
|	엘리먼트 단위 논리 OR
||	논리 OR

할당 연산자

연산자	설명
<-, <<-, =	왼쪽 변수에 할당
->, ->>	오른쪽 변수에 할당

중위 연산자

- R에서 사용하는 대부분의 연산자는 2항 연산자(2 개의 피연산자가 있음)
- 피연산자간에 사용되는 중위 연산자(infix operator)

- 이 연산자는 백그라운드에서 함수 호출을 함
  - 예를 들어, a + b라는 표현식은 실제로 인자 a와 b를 갖는 `+`()함수를 호출
  - `+`(a, b)

```
> 5+3
[1] 8
> `+`(5,3)
[1] 8
> 5-3
[1] 2
> `-`(5,3)
[1] 2
> 5*3-1
[1] 14
>`-`(`*`(5,3),1)
[1] 14
```

중위연산자	설명
%%%	Remainder operator(나머지)
%/%	Integer division(몫)
%*%	Matrix multiplication(행렬의 곱) 또는 벡터의 내적
%o%	Outer product(외적)
%x%	Kronecker product(크로네커 곱)
%in%	Matching operator(매칭 연산자)

함수

- 함수는 코드를 논리적으로 단순한 부분으로 나누어 유지하고 이해하기 쉽게 만든다.
- 함수 정의

```
func_name <- function (argument) {
    statement
}
```

  - 함수를 정의하기 위해 function 예약어를 사용
  - 중괄호({ }) 안에 있는 명령문은 함수 본문을 구성
  - 함수의 본문이 단일 표현식을 경우에는 중괄호를 생략할 수 있다.
  - 함수 객체는 func_name에 할당하여 이름을 부여받는다.
  - 예

```
pow <- function(x, y) {
        result <- x^y
        print(paste0(x, "의 ", y, "승은 ", result, "입니다."))
}
```

- 함수 사용

```
> pow(2,3)
[1] "2의 3승은 8입니다.“

> pow(3,2)
[1] "3의 2승은 9입니다."
```

이름을 갖는 인자

- 함수 호출에서 실제 인수에 대한 형식 인수의 인수 일치는 위치 순서대로 발생
  - pow(2,3)에서 형식 인자 x와 y가 각각 2와 3으로 지정된다는 것을 의미
- R에서는 이름이 지정된 인수를 사용하여 함수를 호출 할 수도 있다.
  - 실제 인수의 순서는 중요하지 않다.

```
> pow(2,3)
[1] "2의 3승은 8입니다.“
> pow(x=2, y=3)
[1] "2의 3승은 8입니다.“
> pow(y=3, x=2)
[1] "2의 3승은 8입니다.“
```

- 명명 된 인수와 명명되지 않은 인수를 사용할 수 있다.
  - 모든 명명 된 인수가 먼저 일치 된 다음 나머지 이름이 없는 인수가 위치 순서대로 매치된다.

```
> pow(x=2, 3)
[1] "2의 3승은 8입니다.“
> pow(3, x=2)
[1] "2의 3승은 8입니다."
```

인자의 기본값

- 인자의 기본값
  - R의 함수에서 인수에 기본값을 할당 할 수 있다.
  - 함수 선언의 형식 인수에 적절한 값을 제공하여 수행된다.
  - 다음 함수는 y에 대한 기본값이 있다.

```
pow <- function(x, y=2) {
 result <- x^y
 print(paste0(x, "의 ", y, "승은 ", result, "입니다."))
}
```

  - 인수에 기본값을 사용하면 함수를 호출 할 때 기본값을 갖는 인수는 선택적이 될 수 있다.

```
> pow(3)
[1] "3의 2승은 9입니다.“
> pow(3,3)
[1] "3의 3승은 27입니다."
```

리턴문

- return(expression)
  - 함수에서 리턴된 값은 유효한 오브젝트가 될 수 있다.

```
check <- function(x) {
        if (x > 0) {
                result <- "Positive"
        }else if (x < 0) {
                result <- "Negative"
        }else {
                result <- "Zero"
        }
        return(result)
}
> check(1)
[1] "Positive"
> check(-3)
[1] "Negative"
> check(0)
[1] "Zero"
```

리턴문

- return()문이 없는 함수
  - 함수에서 명시적으로 반환 값이 없으면 마지막으로 평가 된 표현식의 값이 자동으로 R에 반환.

```
check <- function(x) {
        if (x > 0) {
                result <- "Positive"
        }else if (x < 0) {
                result <- "Negative"
        }else{
                result <- "Zero"
        }
        result
}
```

리턴문

- 다중 리턴
  - return() 함수는 단일 객체 만 반환 할 수 있다.
  - R에 여러 값을 반환하려는 경우 list 객체(또는 다른 객체)를 사용하여 이를 반환 할 수 있다.

```
multi_return <- function() {
        my_list <- list("color" = "red", "size" = 20, "shape" = "round")
        return(my_list)
}
```

가변인자

- 가변인자
  - 예) mean(x, ...)
  - ... 에 의한 추가 인수는 다른 함수로 전달되거나 다른 함수로부터 전달
  -인자로 다양한 값을 전달 할 수 있다. a

```
dd <- function(...) {
        args <- list(...) su
        m <- 0; for(data
        in args) {
                        sum = sum + data;
        }
        print(sum)
}
```

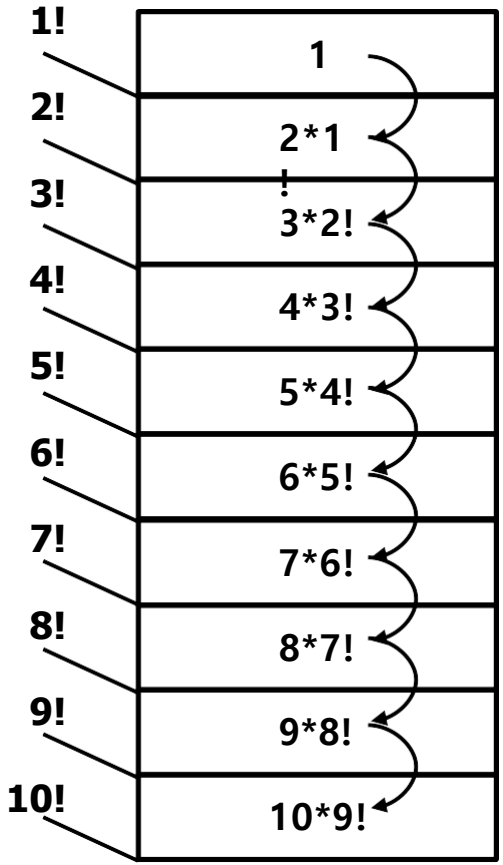
  - 이 add() 함수는 인자에 몇 개를 전달하더라도 함수는 올바른 결과를 출력.

```
> add(1,2)
[1] 3
> add(1,2,3)
[1] 6
>add(1,2,3,4)
[1] 10
```

재귀 호출

- 재귀 호출
  - 자기 자신을 호출하는 함수를 재귀 함수(Recursive Function)
  - 이 프로그래밍 기술은 문제를 더 작고 간단한 하위 문제로 분해하여 해결할 수 있다.

```
recurse <- function() {
        ...
        recurse()
}
```



R 환경(EnvirOnment)

- R 환경(EnvirOnment)은 객체 (함수, 변수 등)의 집합
- R 인터프리터를 시작할 때 환경이 만들어진다.
- R 명령 프롬프트에서 사용할 수 있는 최상위 환경은 R_GlobalEnv라는 글로벌 환경
  - 글로벌 환경은 R 코드에서 .GlobalEnv
- ls() 함수를 사용하여 현재 환경에서 정의 된 변수 및 함수를 표시

```
> rm(list=ls())
> a <- 3
> b <- 7
> f <- function(x) x <- 1
> ls()
```

- [1] "a" "b" "f"

유효 범위

- 변수들은 어디에 선언되어 있는지에 따라 다른 유효 범위(Scope)를 갖는다.

```
outer_func <- function(){
        b <- 20
        inner_func <- function(){
                c <- 30
        }
}
a <- 10
```

- 전역변수(Global Variable)
  - 프로그램을 실행하는 동안 존재하는 변수
  - 프로그램의 어느 부분에서든지 변경하고 액세스 할 수 있다.
  - 그러나 전역 변수는 함수의 관점에도 의존
    - 예를 들어 위의 예에서 inner_func()의 관점에서 a와 b는 모두 전역 변수
    - 그러나 outer_func()의 관점에서 보면 b는 지역 변수이고 a는 전역 변수입니다. 변수 c는 outer_func()에서 완전히 보이지 않는다.
- 지역변수(Local Variable)
  - 지역 변수(Local Variable)는 함수처럼 프로그램의 특정 부분에만 존재하는 변수
  - 수 호출이 끝나면 해제
  - 변수 c는 지역 변수
  - 함수 inner_func()를 사용하여 변수에 값을 할당하면 변경은 로컬 일 뿐이며 함수 외부에서는 액세스 할 수 없다.
  - 전역 변수와 지역 변수의 이름이 일치하는 경우에도 동일합니다.

값에 의한 호출

- R에서 함수 호출방식은 값에 의한 호출
- 함수 밖에서 선언한 함수를 함수 안에서 바꾸더라도 함수 밖의 변수에는 영향을 주지 않는다.

```
a <- 10
 b <- 20
 func <- function(a, b) {
        a <- a+ 10
        b <- b+ 10
        return (a+ b)
 }
```

- 우연의 일치 인 것처럼 전역변수와 지역변수의 이름이 같지만 실제 프로그램에서는 전혀 다른 공간에 변수가 만들어지기 때문에 다른 변수
- 위 함수를 호출하는 구문이 실행된 다음 함수 밖의 a는 바뀌지 않는다.
- 다음 코드는 함수를 호출한 후 a 변수의 값을 출력하는 예. 함수 안에서 a는 20이 되지만 이는 함수 밖의 a변수와는 완전히 다른 변수다.

```
 >func(a,b)
 [1] 50
 > a
 [1] 10
```

전역변수에 값 할당

- 전역 변수를 읽을 수는 있지만 할당하려고 하면 대신 새로운 지역 변수가 생성
- 전역 변수에 할당하려면 슈퍼 할당 연산자 <<-가 사용
- 됩니다. 함수 내에서 이 연산자를 사용하면 상위 환경 프레임에서 변수를 검색
  - 발견되지 않으면 변수가 전역 환경에 도달 할 때까지 계속해서 다음 상위 레벨을 검색
  - 변수가 여전히 발견되지 않으면 변수가 작성되고 전역 레벨로 지정

```
Outer_func <- function(){
        inner_func <- function(){
                a <<- 30
                print(a)
        }
        inner_func()
        print(a)
}
> Outer_func()
[1] 30
[1] 30
>print(a)
[1] 30
```

연습문제

- 소수(Prime Number)체크

  If문과 for문을 이용하여 매개변수가 소수(prime Number)인지 아닌지 TRUE나 FALSE를 return 하는 함수를 작성하고 호출하시오