



## 식과 제어문

- 식
- 조건문
- 반복문
- 무조건 분기문
- 구조적 프로그래밍



## 01\_식

- 식
  - 계산을 표현하는 기본적인 수단
  - 연산자, 피연산자, 괄호, 함수 호출 등으로 구성
- 연산자
  - 한 개의 피연산자를 갖는 단항 연산자
  - 두 개의 피연산자를 갖는 이항 연산자
  - C 기반 언어 삼항 연산자
    - $(i\%2) ? \text{"odd"} : \text{"even"}$
  - 대부분의 프로그래밍 언어에서 이항 연산자는 피연산자들 사이에 위치
    - $x + y$
  - LISP는 연산자가 피연산자들보다 앞에 위치
    - $(+ x y)$

## 01\_식

### ■ 연산자 표기 방법

- 중위 표기법(infix notation) :
  - 연산자가 피연산자들 사이에 위치하는 표기법
- 전위 표기법(prefix notation)
  - 연산자가 피연산자들보다 앞에 위치하는 표기법
    - 예) 함수 호출 표기 : `add(1, mul(2, 3))`
- 후위 표기법(postfix notation)
  - 연산자가 피연산자들보다 뒤에 위치하는 표기법
- 표기법 비교

표기법	예_1	예_2
중위표기법	$2 * 3$	$1 * 2 - 3$
전위표기법	$* 2 3$	$- * 1 2 3$
후위표기법	$2 3 *$	$1 2 * 3 -$

## 01\_식

### ■ 연산자 평가 순서

- 여러 개의 연산자로 이루어진 식에서는 연산자 평가 순서가 중요
  - 어떤 연산자가 먼저 평가되느냐에 따라 연산 결과가 달라짐
- 우선순위, 결합 규칙, 그리고 괄호에 의해 결정
- 예 :  $x = 1, y = 2, z = 3$

$x + y * z$

- 왼쪽에서 오른쪽으로 평가되면 9
- 오른쪽에서 왼쪽으로 평가되면 7
- 평가되는 순서에 따라 전혀 다른 결과

## 01\_식

### ■ 연산자 우선순위

FORTRAN	C	Ada	우선순위
**	후위 ++, --		
*, /	전위 ++, --, !	**	
+, -	*, /, %	*, /, mod, rem	
.EQ., .NE., .LT., .LE., .GT., .G	+, -	+, -, not	
E.	<, <=, >, >=	=, /=, <, <=, >, >=	
.NOT.	==, !=	and, or, not	
.AND.	&&		
.OR.			

- 예 ① :  $x + y * z$ 
  - +의 우선순위 < \*의 우선순위 →  $y*z$ 가 먼저 계산됨
- 예 ② :  $x + y ** z * x$ 
  - \*\*는 지수 연산자로 우선순위가 가장 높음
  - $x=2, y=3, z=4$ 일 때, 결과는 164가 됨

## 01\_식

- 예 ③ :  $x - y + z$ 
  - 왼쪽에 위치한 -가 먼저 평가되면 → 2
  - 오른쪽에 위치한 +가 먼저 평가되면 → -4
  - 같은 우선순위의 연산자라 하더라도 평가되는 위치에 따라 전혀 다른 결과를 가져올 수 있음

## 01\_식

### ■ 결합 규칙

- 좌 결합 규칙 : 왼쪽에서부터 오른쪽으로 평가
- 우 결합 규칙 : 오른쪽에서부터 왼쪽으로 평가
- 예
  - FORTRAN에서 지수 연산자는 우 결합 규칙을 적용  $\rightarrow x ** y ** z$
  - Ada에서 지수 연산자는 결합 규칙을 갖지 않음  $\rightarrow$  괄호 사용 권장

### ■ 괄호 사용

- 우선순위와 결합 규칙에 관계없이 괄호 안의 연산이 먼저 평가
- 순서를 명확히 정해야 함
  - 예:  $(x ** y) ** z$  또는  $x ** (y ** z)$   $x * (y + z)$
- 장점: 모든 연산자의 평가 순서를 괄호로 표현하면 우선순위나 결합 규칙을 기억할 필요가 없음
- 단점: 식의 작성을 지루하게 하고 판독성을 떨어뜨림

## 01\_식

### ■ 피연산자 평가 순서

#### ■ 문제가 있는 C 예제

```

01 #include <stdio.h>
02 int x = 10;
03 int func(void)
04 {
05     x = 20;
06     return 30;
07 }
08 int main(void)
09 {
10     printf("%d\n", x + func());
11     return 0;
12 }

```

- 10행의  $x + func()$ 에서 왼쪽 피연산자인  $x$ 가 먼저 평가

x	+	func()
10		30

- 오른쪽 피연산자인  $func()$  함수가 먼저 평가

x	+	func()
20		30

## 01\_식

- 함수의 부작용
  - 피연산자의 평가 순서에 따라 다른 결과가 나오는 문제를 해결하기 위한 방법
  - 함수에서 부작용을 일으키지 못하도록 하는 것
  - 피연산자의 평가 순서를 정해놓는 것
    - Java(피연산자들의 평가 순서를 왼쪽에서 오른쪽으로 정해놓고 있음)

```

01 public class operand{
02     static int x=10;
03     public static int func(){
04         x = 20;
05         return 30;
06     }
07     public static void main(String[] args){
08         System.out.println(x + func());
09     }
10 }

```

## 01\_식

- 단락 회로 평가
  - 모든 피연산자와 연산자를 평가하지 않고서도 식의 결과가 결정되는 것을 의미
  - 예
    - true or x → x의 값에 관계없이 true
    - false and x → x의 값에 관계없이 false
  - 단락회로 평가를 지원하지 않는다고 가정한 경우의 예제

```

int data[5];
int index=0;
:
while (index <= 4 && data[index]!=key)
    index++;

```

- index가 5가 되어 index<=4가 거짓이 되어도 data[index]!=key를 평가함 → data[5]!=key에 의해 배열 첨자 범위를 이탈

## 01\_식

- 부작용을 포함하는 식의 경우
  - 단락회로 평가를 조심해서 사용
  - 예:  $(x > y) \parallel (x++ \% 2)$ 
    - 두 번째 식은  $x \leq y$ 인 경우에만 평가되므로  $x$  값 역시  $x \leq y$ 인 경우에만 증가
- 각 언어별 단락회로 지원 사항
  - Pascal의 and, or은 단락회로 평가를 지원하지 않음
  - C, C++, Java의 &&, ||는 단락회로 평가를 지원
  - Ada는 단락회로 평가를 지원하는 연산자와 지원하지 않는 연산자를 구분
  - 단락회로 평가를 하는 and then을 이용한 Ada의 예

```
while (index <= lastindex) and then (data(index) /= key) loop
  index = index + 1;
end loop;
```

## 01\_식

- 중복 연산자
  - 하나의 기호가 두 가지 이상의 목적으로 사용되는 연산자

```
10 + 20;
1.2 + 3.14;
"Hello" + "world"
```