연산자

표현식과 연산자

값은 다양한 방법으로 생성할 수 있다. 여기서 말하는 다양한 방법이란 표현식을 말한다. 프로그래밍 언어에서 표현식은 중요한 역할을 한다. 다시 한번 표현식에 대해 살펴보자.

표현식(expression)은 리터럴, 식별자, 연산자, 함수 호출 등의 조합을 말한다. 표현식은 평가(evaluation. 표현식을 실행하여 하나의 값을 만드는 과정)되어 하나의 값을 만든다. 즉, 표현식은 하나의 값으로 평가될 수 있는 문(statement, 문과 표현식 참고)이다.

표현식은 리터럴 표현식, 식별자 표현식, 연산자 표현식, 함수/메소드 호출 표현식 등으로 나누어 볼 수 있지만 결국 평가되어 하나의 값을 만든다는 점에서 모두 동일하다.

```
// 리터럴 표현식
10

// 식별자 표현식
sum

// 연산자 표현식
10 + 20

// 함수/메소드 호출 표현식
square()
```

표현식은 평가되어 결국 하나의 값이 되므로 표현식과 값은 동등한 관계, 즉 동치(Equivalent)다. 다시 말해, 표현식은 값처럼 사용할 수 있다. 이것은 값이 위치할 수 있는 자리에는 표현식도 위치할 수 있다는 의미다. 예를 들어 산술 연산자 +의 좌항과 우항에는

숫자 값이 위치해야 한다. 숫자 값으로 평가될 수 있는 표현식이라면 숫자 값 대신 사용할 수 있다.

```
var x = 10;

// 연산자 표현식
x + 30;  // 식별자 표현식과 숫자 리터럴과 연산자의 조합
```

이처럼 표현식은 다른 표현식의 일부가 되어 새로운 값을 만들어낼 수 있다. 연산자 표현식은 표현식을 결합해 새로운 값을 만들어 내는 가장 일반적인 표현식이다.

문과 표현식

문(statement)은 자바스크립트 엔진에게 내리는 명령이다. 문이 실행되면 무슨 일인가가 일어나게 된다. 변수 선언문을 실행하면 변수가 선언이 되고, 할당문을 실행하면 할당이 된다. 조건문을 실행하면 주어진 조건에 따라 코드 블록({...})의 실행이 결정되고, 반복문을 실행하면 코드 블록이 반복 실행된다.

```
// 변수 선언문
var x;

// 할당문
x = 5;

// 함수 선언문
function foo () {}

// 조건문
if (x > 5) { … }

// 반복문
for (var i = 0; i < 10; i++) { … }
```

문은 리터럴, 연산자, 표현식, 키워드 등으로 구성되며 세미콜론(;)으로 끝나야 한다. (코드 블록 { ... }은 제외)

문의 끝에 붙이는 세미콜론은 옵션으로 쓰지 않아도 상관없다. 자바스크립트 엔진이 스크립트를 해석할 때, 자바스크립트 엔진에는 문의 끝이라고 예측되는 지점에 세미콜론을 자동으로 붙여주는 세미콜론 자동 삽입 기능(ASI, automatic semicolon insertion)이 있기 때문이다. 하지만 세미콜론 자동 삽입 기능의 예측과 개발자의 예측과 다른 경우가 간혹 있다.

```
function foo () {
  return
    {}
}
console.log(foo()); // undefined
```

세미콜론을 반드시 붙여야 한다는 주장이 대다수를 차지하지만 붙이지 말아야 한다 주장도 설득력이 있다. 하지만 ESLint와 같은 정적 분석 도구에서도 세미콜론 사용을 기본으로 설정하고 있고 TC39(ECMAScript 기술 위원회)도 세미콜론 사용을 권장하는 분위기이므로 세미콜론을 붙이도록 하겠다.

자바스크립트의 모든 코드는 문 또는 표현식이다. 자연어에서 문이 마침표로 끝나는 하나의 완전한 문장(Sentence)이라고 한다면 표현식은 문을 구성하는 요소이다. 표현식은 그 자체로 하나의 문이 될 수도 있다.

표현식과 문은 유사하여 구별이 어려울 수 있다. 표현식은 평가되어 값을 만들지만 그 이상의 행위는 할 수 없다. 문은 var, let, const, function, class와 같은 선언 키워드를 사용하여 변수나 함수를 생성하기도 하고 if, for, while 문과 같은 제어문을 생성하여 프로그램의 흐름을 제어하기도 한다.

표현식의 역할은 값을 생성하는 것이다. 문의 역할은 표현식으로 생성한 값을 사용해 컴퓨터에게 명령을 내리는 것이다. 문에는 표현식인 문과 표현식이 아닌 문이 있다. 예를 들어 선언문은 값으로 평가될 수 없다. 따라서 표현식이 아닌 문이다. 하지만 할당문은 그 자체가 표현식인 문이다. 아래 예제를 살펴보자.

```
// 선언문(Declaration statement)
var x = 5 * 10; // 표현식 x = 5 * 10를 포함하는 문이다.

// 할당문(Assignment statement)
x = 100; // 이 자체가 표현식이지만 완전한 문이기도 하다.
```

위 예제의 선언문은 표현식이 아닌 문이다. 다시 말해 값으로 평가될 수 없다. 따라서 선언문은 값처럼 사용할 수 없다.

```
var foo = var x = 5 * 10;
```

이에 반해 할당문은 그 자체가 표현식이다. 다시 말해 할당문은 표현식인 문이다.

> 크롬 DevTools에서 표현식이 아닌 문은 언제나 undefined를 반환하고, 표현식인 문은 언제나 값을 반환한다.

할당문은 표현식인 문이기 때문에 값처럼 사용할 수 있다.

```
var foo = x = 100;
```

할당문을 값처럼 변수에 할당했다. 할당문은 할당한 값으로 평가된다. 즉, `x = 100`은 변수 x에 할당한 값 100으로 평가될 수 있다. 따라서 변수 foo에는 100이 할당된다.

연산자란?

연산자(Operator)는 하나 이상의 표현식을 대상으로 산술, 할당, 비교, 논리, 타입 연산 등을 수행해 하나의 값을 만든다. 이때 연산의 대상을 피연산자(Operand)라 한다. 피연산자도 평

가되어 하나의 값이 되므로 표현식이고 피연산자를 연산자와 결합한 연산자 표현식도 물론 표현식이다.

```
// 산술 연산자
5 * 4 // 20

// 문자열 연결 연산자
'My name is ' + 'Lee' // "My name is Lee"

// 할당 연산자
var color = 'red'; // "red"

// 비교 연산자
3 > 5 // false

// 논리 연산자
(5 > 3) && (2 < 4)  // true

// 타입 연산자
typeof 'Hi' // "string"
```

피연산자가 “값”이라는 명사의 역할을 한다면 연산자는 “값을 만든다”라는 동사의 역할을 한다고 볼 수 있다. 다시 말해, 피연산자는 연산의 대상이 되어야 하므로 값으로 평가할 수 있어야 한다. 연산자는 값으로 평가된 피연산자를 연산해 새로운 값을 만든다.

자바스크립트가 제공하는 다양한 연산자에 대해 살펴보도록 하자.

산술 연산자

산술 연산자(Arithmetic Operator)는 피연산자를 대상으로 수학적 계산을 수행해 새로운 숫자 값을 만든다. 산술 연산을 할 수 없는 경우에는 NaN을 반환한다.

산술 연산자는 이항 산술 연산자와 단항 산술 연산자로 구분할 수 있다.

이항 산술 연산자

이항 산술 연산자는 2개의 피연산자를 대상으로 연산하여 숫자 타입의 값을 만든다.

모든 이항 산술 연산자는 피연산자의 값을 변경하는 부수 효과(Side effect)가 없다. 다시 말해, 어떤 산술 연산을 해도 피연산자의 값이 바뀌는 경우는 없고 단지 새로운 값을 만들 뿐이다.

이항 산술 연산자	의미
+	덧셈
-	뺄셈
*	곱셈
/	나눗셈
%	나머지

```
5 + 2  // 7
5 - 2  // 3
5 * 2  // 10
5 / 2  // 2.5
5 % 2  // 1
```

단항 산술 연산자

단항 산술 연산자는 1개의 피연산자를 대상으로 연산한다.

증가/감소(++/–) 연산자는 피연산자의 값을 변경하는 부수 효과가 있다. 다시 말해, 증가/감소 연산을 하면 피연산자의 값이 바뀐다.

단항 산술 연산자	의미
++	증가
--	감소
+	어떠한 효과도 없다. 음수를 양수로 반전하지도 않는다.
-	양수를 음수로 음수를 양수로 반전한 값을 반환한다.

증가/감소(++/–) 연산자는 위치에 의미가 있다. 피연산자 앞에 위치한 전위 증가/감소 연산자(Prefix increment/decrement operator)는 먼저 피연산자의 값을 증가/감소시킨 후, 다른 연산을 수행한다. 피연산자 뒤에 위치한 후위 증가/감소 연산자(Postfix increment/decrement operator)는 먼저 다른 연산을 수행한 후, 피연산자의 값을 증가/감소시킨다.

```
var x = 5, result;

// 선대입 후증가 (Postfix increment operator)
result = x++;
console.log(result, x); // 5 6

// 선증가 후대입 (Prefix increment operator)
result = ++x;
console.log(result, x); // 7 7

// 선대입 후감소 (Postfix decrement operator)
result = x--;
console.log(result, x); // 7 6

// 선감소 후대입 (Prefix decrement operator)
result = --x;
console.log(result, x); // 5 5
```

`+` 단항 연산자는 피연산자에 어떠한 효과도 없다. 음수를 양수로 반전하지도 않는다. 그런데 숫자 타입이 아닌 피연산자에 사용하면 피연산자를 숫자 타입으로 변환하여 반환한다. 이

때 피연산자를 변경하는 것은 아니고 숫자 타입으로 변환한 값을 생성해서 반환한다. 따라서 부수 효과는 없다.

```
+10 // 10
+'10' // 10
+true // 1
+false // 0
```

`-` 단항 연산자는 피연산자의 부호를 반전한 값을 반환한다. + 단항 연산자와 마찬가지로 숫자 타입이 아닌 피연산자에 사용하면 피연산자를 숫자 타입으로 변환하여 반환한다. 이때 피연산자를 변경하는 것은 아니고 부호를 반전한 값을 생성해서 반환한다. 따라서 부수 효과는 없다.

```
-10 // -10
-'10' // -10
-true // -1
-false // -0
```

문자열 연결 연산자

`+` 연산자는 피연산자 중 하나 이상이 문자열인 경우 문자열 연결 연산자로 동작한다. 그 외의 경우는 덧셈 연산자로 동작한다. 아래 예제를 살펴보자.

```
// 문자열 연결 연산자
'1' + '2'      // '12'
'1' + 2       // '12'

// 산술 연산자
1 + 2          // 3
1 + true       // 2 (true → 1)
1 + false      // 1 (false → 0)
true + false    // 1 (true → 1 / false → 0)
```

```
1 + null       // 1 (null → 0)
1 + undefined // NaN (undefined → NaN)
```

이 예제에서 주목할 것은 개발자의 의도와는 상관없이 자바스크립트 엔진에 의해 암묵적으로 타입이 자동 변환되기도 한다는 것이다. 위 예제에서 1 + true를 연산하면 자바스크립트 엔진은 암묵적으로 불리언 타입의 값인 true를 숫자 타입인 1로 타입을 강제 변환한 후 연산을 수행한다. 이를 암묵적 타입 변환(Implicit coercion) 또는 타입 강제 변환(Type coercion)이라고 한다. 앞서 살펴본 +/- 단항 연산자도 암묵적 타입 변환이 발생한 것이다. 이에 대해서는 타입 변환에서 자세히 살펴볼 것이다.

할당 연산자

할당 연산자(Assignment Operator)는 우항에 있는 피연산자의 평가 결과를 좌항에 있는 변수에 할당한다. 할당 연산자는 좌항의 변수에 값을 할당하므로 부수 효과가 있다.

할당 연산자	사례	동일 표현
=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y

```
var x;

x = 10;   // 10
x += 5;   // 15
x -= 5;   // 10
x *= 5;   // 50
x /= 5;   // 10
x %= 5;   // 0
```

```
var str = 'My name is ';
str += 'Lee'; // My name is Lee
```

표현식은 "하나의 값으로 평가된다"고 하였다. 그렇다면 할당 연산은 표현식일까? 아래의 예제를 살펴보자.

```
var x;
console.log(x = 10); // 10
```

할당 연산은 변수에 값을 할당하는 부수 효과만 있을 뿐 값으로 평가되지 않을 것처럼 보인다. 하지만 할당 연산은 하나의 값으로 평가되는 표현식이다. 할당 표현식은 할당된 값으로 평가된다. 위 예제의 경우 x에 할당된 숫자 값 10으로 평가된다. 따라서 아래와 같이 할당 연산 표현식을 다른 변수에 할당할 수도 있다.

```
var x, y;
y = x = 10; // 연쇄 할당(Chained assignment)
console.log(x, y); // 10 10
```

비교 연산자

비교 연산자(Comparison Operator)는 좌항과 우항의 피연산자를 비교하여 불리언 값을 반환한다. if 문이나 for 문과 같은 제어문의 조건식에서 주로 사용한다.

동등 / 일치 비교 연산자

비교 연산자	의미	사례	설명
==	동등 비교	x == y	x와 y의 값이 같음
===	일치 비교	x === y	x와 y의 값과 타입이 같음
!=	부등 비교	x != y	x와 y의 값이 다름

비교 연산자	의미	사례	설명
!==	불일치 비교	x !== y	x와 y의 값과 타입이 다름

문자열 연결 연산자에서 언급했듯이 개발자의 의도와는 상관없이 자바스크립트 엔진에 의해 암묵적으로 타입이 자동 변환되기도 한다. 이를 암묵적 타입 변환이라 부른다고 했다.

동등 비교(==) 연산자는 좌항과 우항의 피연산자를 비교할 때 암묵적 타입 변환을 통해 타입을 일치시킨 후 같은 값을 갖는지 비교한다. 따라서 동등 비교 연산자는 좌항과 우항의 피연산자가 타입은 다르더라도 암묵적 타입 변환 후에 같은 값을 수 있으면 true를 반환한다.

```
// 동등 비교
5 == 5    // true
// 타입은 다르지만 암묵적 타입 변환을 통해 타입을 일치시키면 같은 값을 같는
5 == '5'   //true
5 == 8    // false
```

결론부터 말하자면 동등 비교 연산자는 편리한 경우도 있지만 수많은 부작용을 일으키므로 사용하지 않는 편이 좋다. 아래 예제를 살펴보자.

```
'' == '0'           // false
0 == ''             // true
0 == '0'            // true

false == 'false'    // false
false == '0'        // true

false == undefined  // false
false == null       // false
null == undefined   // true
```

위 예제와 같은 코드를 작성할 개발자는 드물겠지만 이처럼 동등 비교(==) 연산자는 예측하기 어려운 결과를 만들어낸다. 위 예제는 이해하려 하지 않아도 된다. 다만 동등 비교 연산자

를 사용하지 말고 일치 비교 연산자를 사용하면 된다.

일치 비교(===) 연산자는 좌항과 우항의 피연산자가 타입도 같고 값도 같은 경우에 한하여 true를 반환한다.

```
// 일치 비교
5 === 5   // true
5 === '5' // false
```

일치 비교 연산자는 예측하기 쉽다. 위에 살펴본 동등 비교 연산자의 해괴망측한 예제는 모두 false를 반환한다. 일치 비교 연산자에서 주의할 것은 NaN이다.

```
NaN === NaN // false
```

NaN은 자신과 일치하지 않는 유일한 값이다. 따라서 숫자가 NaN인지 조사하려면 빌트인 함수 isNaN을 사용한다

```
isNaN(NaN) // true
```

숫자 0도 주의하도록 하자.

```
0 === -0     // true
```

부동등 비교 연산자(!=)와 불일치 비교 연산자(!==)는 동등 비교(==) 연산자와 일치 비교(===) 연산자의 반대 개념이다.

```
// 부동등 비교
5 != 8    // true
5 != 5    // false
5 != '5'  // false

// 불일치 비교
5 !== 8   // true
5 !== 5   // false
5 !== '5' // true
```

대소 관계 비교 연산자

대소 관계 비교 연산자는 피연산자의 크기를 비교하여 불리언 값을 반환한다.

대소 관계 비교 연산자	예제	설명
>	x > y	x가 y보다 크다
<	x < y	x가 y보다 작다
>=	x >= y	x가 y보다 같거나 크다
<=	x <= y	x가 y보다 같거나 크다

```
// 대소 관계 비교
5 > 0    // true
5 > 5    // false
5 > 8    // false

5 < 0    // false
5 < 5    // false
5 < 8    // true

5 >= 0   // true
5 >= 5   // true
5 >= 8   // false

5 <= 0   // false
```

```
5 <= 5   // true
5 <= 8   // true
```

삼항 조건 연산자

삼항 조건 연산자(ternary operator)는 조건식의 평가 결과에 따라 반환할 값을 결정한다. 자바스크립트의 유일한 삼항 연산자이며 부수 효과는 없다. 삼항 조건 연산자 표현식은 아래와 같이 사용한다.

```
조건식 ? 조건식이 ture일때 반환할 값 : 조건식이 false일때 반환할 값
```

물음표(?) 앞의 첫번째 피연산자가 조건식, 즉 불리언 타입의 값으로 평가될 표현식이다. 만약 조건식의 평가 결과가 불리언 값이 아니면 불리언 값으로 암묵적 타입 변환된다. 이때 조건식이 참이면 콜론(:) 앞의 두번째 피연산자가 평가되어 반환되고, 거짓이면 콜론(:) 뒤의 세번째 피연산자가 평가되어 반환된다.

```
var x = 2;

// x가 짝수이면 '짝수'를 홀수이면 '홀수'를 반환한다.
// 2 % 2는 0이고 0은 false로 암묵적 타입 변환된다.
var result = x % 2 ? '홀수' : '짝수';

console.log(result); // 짝수
```

삼항 조건 연산자는 다음 장에서 살펴볼 if...else 문을 사용해도 동일한 처리를 할 수 있다.

```
var x = 2, result;

// x가 짝수이면 '짝수'를 홀수이면 '홀수'를 반환한다.
// 2 % 2는 0이고 0은 false로 암묵적 타입 변환된다.
```

```
if (x % 2) result = '홀수';
else       result = '짝수';

console.log(result); // 짝수
```

하지만 if...else 문은 표현식이 아닌 문이다. 따라서 if...else 문은 값으로 평가할 수 없다. 하지만 삼항 조건 연산자 표현식은 값으로 평가할 수 있는 표현식이다. 따라서 삼항 조건 연산자식은 다른 표현식의 일부가 될 수 있어 매우 유용하다.

논리 연산자

논리 연산자(Logical Operator)는 우항과 좌항의 피연산자(부정 논리 연산자의 경우, 우항의 피연산자)를 논리 연산한다.

논리 부정(!) 연산자는 언제나 불리언 값을 반환한다. 하지만 논리합(`||`) 연산자와 논리곱(`&&`) 연산자는 일반적으로 불리언 값을 반환하지만 반드시 불리언 값을 반환해야 하는 것은 아니다.

논리 연산자	의미
||	논리합(OR)
&&	논리곱(AND)
!	부정(NOT)

```
// 논리합(||) 연산자
true || true   // true
true || false  // true
false || true  // true
false || false // false

// 논리곱(&&) 연산자
true && true   // true
true && false  // false
false && true  // false
false && false // false
```

```
// 논리 부정(!) 연산자
!true  // false
!false // true
```

논리 부정(!) 연산자는 언제나 불리언 값을 반환한다. 단, 피연산자는 반드시 불리언 값일 필요는 없다. 만약 피연산자가 불리언 값이 아니면 불리언 타입으로 암묵적 타입 변환된다.

```
// 암묵적 타입 변환
!0 // true
```

하지만 논리합(`||`) 연산자와 논리곱(`&&`) 연산자의 연산 결과는 불리언 값이 아닐 수도 있다. 이 두 연산자는 언제나 피연산자 중 어는 한쪽 값을 반환한다.

```
// 단축 평가
'Cat' && 'Dog' // “Dog”
```

쉼표 연산자

쉼표(,) 연산자는 왼쪽 피연산자부터 차례대로 피연산자를 평가하고 마지막 피연산자의 평가가 끝나면 마지막 피연산자의 평가 결과를 반환한다.

```
var x, y, z;
x = 1, y = 2, z = 3; // 3
```

그룹 연산자

그룹 ((...))연산자는 그룹 내의 표현식을 최우선으로 평가한다. 그룹 연산자를 사용하면 연산자의 우선 순위를 1순위로 높일 수 있다.

```
10 * 2 + 3   // 23
10 * (2 + 3) // 50
```

typeof 연산자

typeof 연산자는 자신의 뒤에 위치한 피연산자의 데이터 타입을 문자열로 반환한다. typeof 연산자가 반환하는 문자열은 7개의 데이터 타입과 일치하지는 않는다. typeof 연산자는 7가지 문자열 “string”, “number”, “boolean”, “undefined”, “symbol”, “object”, “function” 중 하나를 반환한다. “null”을 반환하는 경우는 없으며 함수의 경우 “function”을 반환한다.

```
typeof ''              // "string"
typeof 1               // "number"
typeof NaN             // "number"
typeof true            // "boolean"
typeof undefined       // "undefined"
typeof Symbol()        // "symbol"
typeof null            // "object"
typeof []              // "object"
typeof {}              // "object"
typeof new Date()      // "object"
typeof /test/gi        // "object"
typeof function () {}  // "function"
```

주의해야 할 것은 typeof 연산자로 null 값을 연산해 보면 null이 아닌 “object”를 반환한다는 것이다. 이것은 자바스크립트의 첫 번째 버전에서 이렇게 설계된 것을 현재의 버전에 반영하지 못하고 있기 때문이다.

```
typeof null  // "object"
```

따라서 null 타입을 확인할 때는 typeof 연산자를 사용하지 말고 일치 연산자(===)를 사용하도록한다.

```
var foo = null;
console.log(typeof foo === null); // false
console.log(foo === null);        // true
```

또 하나 주의해야 할 것이 있다. 선언하지 않은 식별자를 typeof 연산자로 연산해 보면 ReferenceError가 발생하지 않고 “undefined”를 반환한다.

```
typeof undeclared  // "undefined"
```

typeof 연산자가 선언하지 않은 식별자를 연산했을 때 “undefined”를 반환하는 것을 카일 심슨의 “You don’t know JS”에서는 특별한 안전 가드(safety guard)로 설명한다. 하지만 모던 자바스크립트 개발에서는 대부분 모듈을 사용하고 전역 변수인 플래그를 사용하지 않으므로 의도적으로 사용할 필요는 없다.