Chapter 25: Advanced Operators

Section 25.1: Bitwise Operators

Swift Bitwise operators allow you to perform operations on the binary form of numbers. You can specify a binary literal by prefixing the number with `0b`, so for example `0b110` is equivalent to the binary number 110 (the decimal number 6). Each 1 or 0 is a bit in the number.

Bitwise NOT `~`:

```
var number: UInt8 = 0b01101100
let newNumber = ~number
// newNumber is equal to 0b01101100
```

Here, each bit get changed to its opposite. Declaring the number as explicitly `UInt8` ensures that the number is positive (so that we don't have to deal with negatives in the example) and that it is only 8 bits. If `0b01101100` was a larger UInt, there would be leading 0s that would be converted to 1s and become significant upon inversion:

```
var number: UInt16 = 0b01101100
// number equals 0b0000000001101100
// the 0s are not significant
let newNumber = ~number
// newNumber equals 0b1111111110010011
// the 1s are now significant
```

- 0 -> 1
- 1 -> 0

Bitwise AND `&`:

```
var number = 0b0110
let newNumber = number & 0b1010
// newNumber is equal to 0b0010
```

Here, a given bit will be 1 if and only if the binary numbers on both sides of the `&` operator contained a 1 at that bit location.

- 0 & 0 -> 0
- 0 & 1 -> 0
- 1 & 1 -> 1

Bitwise OR `|`:

```
var number = 0b0110
let newNumber = number | 0b1000
// newNumber is equal to 0b1110
```

Here, a given bit will be 1 if and only if the binary number on at least one side of the `|` operator contained a 1 at that bit location.

- 0 | 0 -> 0
- 0 | 1 -> 1
- 1 | 1 -> 1

Bitwise XOR (Exclusive OR) `^`:

```
var number = 0b0110
let newNumber = number ^ 0b1010
// newNumber is equal to 0b1100
```

Here, a given bit will be 1 if and only if the bits in that position of the two operands are different.

- 0 ^ 0 -> 0
- 0 ^ 1 -> 1
- 1 ^ 1 -> 0

For all binary operations, the order of the operands makes no difference on the result.

Section 25.2: Custom Operators

Swift supports the creation of custom operators. New operators are declared at a global level using the `operator` keyword.

The operator's structure is defined by three parts: operand placement, precedence, and associativity.

1. The `prefix`, `infix` and `postfix` modifiers are used to start an custom operator declaration. The `prefix` and `postfix` modifiers declare whether the operator must be before or after, respectively, the value on which it acts. Such operators are urnary, like `8` and `3++` `**`, since they can only act on one target. The `infix` declares a binary operator, which acts on the two values it is between, such as `2+3`.
2. Operators with higher **precedence** are calculated first. The default operator precedence is just higher than `?...:` (a value of 100 in Swift 2.x). The precedence of standard Swift operators can be found here.
3. **Associativity** defines the order of operations between operators of the same precedence. Left associative operators are calculated from left to right (reading order, like most operators), while right associative operators calculate from right to left.

`Version ≥ 3.0`

Starting from Swift 3.0, one would define the precedence and associativity in a **precedence group** instead of the operator itself, so that multiple operators can easily share the same precedence without referring to the cryptic numbers. The list of standard precedence groups is shown below.

Operators return values based on the calculation code. This code acts as a normal function, with parameters specifying the type of input and the `return` keyword specifying the calculated value that the operator returns.

Here is the definition of a simple exponential operator, since standard Swift does not have an exponential operator.

```
import Foundation

infix operator ** { associativity left precedence 170 }

func ** (num: Double, power: Double) -> Double{
    return pow(num, power)
}
```

The `infix` says that the `**` operator works in between two values, such as `9**2`. Because the function has left associativity, `3**3**2` is calculated as `(3**3)**2`. The precedence of `170` is higher than all standard Swift operations, meaning that `3+2**4` calculates to `19`, despite the left associativity of `**`.

`Version ≥ 3.0`

```
import Foundation

infix operator **: BitwiseShiftPrecedence

func ** (num: Double, power: Double) -> Double {
    return pow(num, power)
}
```

Instead of specifying the precedence and associativity explicitly, on Swift 3.0 we could use the built-in precedence group BitwiseShiftPrecedence that gives the correct values (same as `<<`, `>>`).

**: The increment and decrement are deprecated and will be removed in Swift 3.

Section 25.3: Overflow Operators

Overflow refers to what happens when an operation would result in a number that is either larger or smaller than the designated amount of bits for that number may hold.

Due to the way binary arithmetic works, after a number becomes too large for its bits, the number overflows down to the smallest possible number (for the bit size) and then continues counting up from there. Similarly, when a number becomes too small, it underflows up to the largest possible number (for its bit size) and continues counting down from there.

Because this behavior is not often desired and can lead to serious security issues, the Swift arithmetic operators `+`, `-`, and `*` will throw errors when an operation would cause an overflow or underflow. To explicitly allow overflow and underflow, use `&+`, `&-`, and `&*` instead.

```
var almostTooLarge = Int.max
almostTooLarge + 1 // not allowed
almostTooLarge &+ 1 // allowed, but result will be the value of Int.min
```

Section 25.4: Commutative Operators

Let's add a custom operator to multiply a CGSize

```
func *(lhs: CGFloat, rhs: CGSize) -> CGSize{
    let height = lhs*rhs.height
    let width = lhs*rhs.width
    return CGSize(width: width, height: height)
}
```

Now this works

```
let sizeA = CGSize(height:100, width:200)
let sizeB = 1.1 * sizeA         //=> (height: 110, width: 220)
```

But if we try to do the operation in reverse, we get an error

```
let sizeC = sizeB * 20          // ERROR
```

But it's simple enough to add:

```
func *(lhs: CGSize, rhs: CGFloat) -> CGSize{
    return rhs*lhs
```

```
}
```

Now the operator is commutative.

```
let sizeA = CGSize(height:100, width:200)
let sizeB = sizeA * 1.1              //=> (height: 110, width: 220)
```

Section 25.5: Overloading + for Dictionaries

As there is currently no simple way of combining dictionaries in Swift, it can be useful to overload the `+` and `+=` operators in order to add this functionality using generics.

```
// Combines two dictionaries together. If both dictionaries contain
// the same key, the value of the right hand side dictionary is used.
func +<K, V>(lhs: [K : V], rhs: [K : V]) -> [K : V] {
    var combined = lhs
    for (key, value) in rhs {
        combined[key] = value
    }
    return combined
}

// The mutable variant of the + overload, allowing a dictionary
// to be appended to 'in-place'.
func +=<K, V>(inout lhs: [K : V], rhs: [K : V]) {
    for (key, value) in rhs {
        lhs[key] = value
    }
}
```

Version ≥ 3.0

As of Swift 3, `inout` should be placed before the argument type.

```
func +=<K, V>(lhs: inout [K : V], rhs: [K : V]) { ... }
```

Example usage:

```
let firstDict = ["hello" : "world"]
let secondDict = ["world" : "hello"]
var thirdDict = firstDict + secondDict // ["hello": "world", "world": "hello"]

thirdDict += ["hello":"bar", "baz":"qux"] // ["hello": "bar", "baz": "qux", "world": "hello"]
```

Section 25.6: Precedence of standard Swift operators

Operators that bound tighter (higher precedence) are listed first.

Operators	Precedence group (≥3.0)	Precedence	Associativity
`.`		∞	left
`?`, `!`, `++`, `--`, `[ ]`, `( )`, `{}`	(postfix)		
`!`, `~`, `+`, `-`, `++`, `--`	(prefix)		
`~>` (swift ≤2.3)		255	left
`<<`, `>>`	BitwiseShiftPrecedence	160	none
`*`, `/`, `%`, `&`, `&*`	MultiplicationPrecedence	150	left
`+`, `-`, `|`, `^`, `&+`, `&-`	AdditionPrecedence	140	left

`...`, `..<`	RangeFormationPrecedence	135	none
`is`, `as`, `as?`, `as!`	CastingPrecedence	132	left
`??`	NilCoalescingPrecedence	131	right
`<`, `<=`, `>`, `>=`, `==`, `!=`, `===`, `!==`, `~=`	ComparisonPrecedence	130	none
`&&`	LogicalConjunctionPrecedence	120	left
`||`	LogicalDisjunctionPrecedence	110	left
	DefaultPrecedence*		none
`?...:`	TernaryPrecedence	100	right
`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `&=`, `|=`, `^=`	AssignmentPrecedence	90	right, assignment
`->`	FunctionArrowPrecedence		right

`Version ≥ 3.0`

- The `DefaultPrecedence` precedence group is higher than `TernaryPrecedence`, but is unordered with the rest of the operators. Other than this group, the rest of the precedences are linear.

- This table can be also be found on Apple's API reference
- The actual definition of the precedence groups can be found in the source code on GitHub