Swift

Expressions

In Swift, there are four kinds of expressions: prefix expressions, binary expressions, primary expressions, and postfix expressions. Evaluating an expression returns a value, causes a side effect, or both.

Prefix and binary expressions let you apply operators to smaller expressions. Primary expressions are conceptually the simplest kind of expression, and they provide a way to access values. Postfix expressions, like prefix and binary expressions, let you build up more complex expressions using postfixes such as function calls and member access. Each kind of expression is described in detail in the sections below.

GRAMMAR OF AN EXPRESSION

expression → try-operator*opt* prefix-expression binary-expressions*opt*
expression-list → expression | expression **,** expression-list

Prefix Expressions

Prefix expressions combine an optional prefix operator with an expression. Prefix operators take one argument, the expression that follows them.

For information about the behavior of these operators, see Basic Operators and Advanced Operators.

For information about the operators provided by the Swift standard library, see Operator Declarations.

In addition to the standard library operators, you use `&` immediately before the name of a variable that's being passed as an in-out argument to a function call expression. For more information and to see an example, see In-Out Parameters.

GRAMMAR OF A PREFIX EXPRESSION

prefix-expression → prefix-operator$_{opt}$ postfix-expression
prefix-expression → in-out-expression
in-out-expression → **&** identifier

Try Operator

A *try expression* consists of the `try` operator followed by an expression that can throw an error. It has the following form:

```
try expression
```

An *optional-try expression* consists of the `try?` operator followed by an expression that can throw an error. It has the following form:

```
try? expression
```

If the *expression* does not throw an error, the value of the optional-try expression is an optional containing the value of the *expression*. Otherwise, the value of the optional-try expression is `nil`.

A *forced-try expression* consists of the `try!` operator followed by an expression that can throw an error. It has the following form:

```
try! expression
```

If the *expression* throws an error, a runtime error is produced.

When the expression on the left-hand side of a binary operator is marked with `try`, `try?`, or `try!`, that operator applies to the whole binary expression. That said, you can use parentheses to be explicit about the scope of the operator's application.

```
sum = try someThrowingFunction() + anotherThrowingFunction()
  // try applies to both function calls
```

```
sum = try (someThrowingFunction() +
  anotherThrowingFunction()) // try applies to both function
  calls
sum = (try someThrowingFunction()) +
  anotherThrowingFunction() // Error: try applies only to the
  first function call
```

A `try` expression can't appear on the right-hand side of a binary operator, unless the binary operator is the assignment operator or the `try` expression is enclosed in parentheses.

For more information and to see examples of how to use `try`, `try?`, and `try!`, see Error Handling.

GRAMMAR OF A TRY EXPRESSION

try-operator → **try** | **try ?** | **try !**

Binary Expressions

Binary expressions combine an infix binary operator with the expression that it takes as its left-hand and right-hand arguments. It has the following form:

```
left-hand argument operator right-hand argument
```

For information about the behavior of these operators, see Basic Operators and Advanced Operators.

For information about the operators provided by the Swift standard library, see Operator Declarations.

NOTE

At parse time, an expression made up of binary operators is represented as a flat list. This list is transformed into a tree by applying operator precedence. For example, the expression `2 + 3 * 5` is initially understood as a flat list of five items, `2`, `+`, `3`, `*`, and `5`. This process transforms it into the tree (2 + (3 * 5)).

GRAMMAR OF A BINARY EXPRESSION

binary-expression → binary-operator prefix-expression
binary-expression → assignment-operator try-operator$_{opt}$ prefix-expression
binary-expression → conditional-operator try-operator$_{opt}$ prefix-expression
binary-expression → type-casting-operator
binary-expressions → binary-expression binary-expressions$_{opt}$

Assignment Operator

The *assignment operator* sets a new value for a given expression. It has the following form:

```
expression = value
```

The value of the *expression* is set to the value obtained by evaluating the *value*. If the *expression* is a tuple, the *value* must be a tuple with the same number of elements. (Nested tuples are allowed.) Assignment is performed from each part of the *value* to the corresponding part of the *expression*. For example:

```
(a, _, (b, c)) = ("test", 9.45, (12, 3))
// a is "test", b is 12, c is 3, and 9.45 is ignored
```

The assignment operator does not return any value.

GRAMMAR OF AN ASSIGNMENT OPERATOR

assignment-operator → **=**

Ternary Conditional Operator

The *ternary conditional operator* evaluates to one of two given values based on the value of a condition. It has the following form:

```
condition ? expression used if true :
   expression used if false
```

If the *condition* evaluates to `true`, the conditional operator evaluates the first expression and returns its value. Otherwise, it evaluates the second expression and returns its value. The unused expression is not evaluated.

For an example that uses the ternary conditional operator, see Ternary Conditional Operator.

GRAMMAR OF A CONDITIONAL OPERATOR

conditional-operator → **?** expression **:**

Type-Casting Operators

There are four type-casting operators: the `is` operator, the `as` operator, the `as?` operator, and the `as!` operator.

They have the following form:

```
expression is type
expression as type
expression as? type
expression as! type
```

The `is` operator checks at runtime whether the *expression* can be cast to the specified *type*. It returns `true` if the *expression* can be cast to the specified *type*; otherwise, it returns `false`.

The `as` operator performs a cast when it is known at compile time that the cast always succeeds, such as upcasting or bridging. Upcasting lets you use an expression as an instance of its type's supertype, without using an intermediate variable. The following approaches are equivalent:

```
func f(_ any: Any) { print("Function for Any") }
func f(_ int: Int) { print("Function for Int") }
let x = 10
f(x)
// Prints "Function for Int"

let y: Any = x
f(y)
// Prints "Function for Any"

f(x as Any)
```

```
// Prints "Function for Any"
```

Bridging lets you use an expression of a Swift standard library type such as `String` as its corresponding Foundation type such as `NSString` without needing to create a new instance. For more information on bridging, see Working with Foundation Types.

The `as?` operator performs a conditional cast of the *expression* to the specified *type*. The `as?` operator returns an optional of the specified *type*. At runtime, if the cast succeeds, the value of *expression* is wrapped in an optional and returned; otherwise, the value returned is `nil`. If casting to the specified *type* is guaranteed to fail or is guaranteed to succeed, a compile-time error is raised.

The `as!` operator performs a forced cast of the *expression* to the specified *type*. The `as!` operator returns a value of the specified *type*, not an optional type. If the cast fails, a runtime error is raised. The behavior of `x as! T` is the same as the behavior of `(x as? T)!`.

For more information about type casting and to see examples that use the type-casting operators, see Type Casting.

GRAMMAR OF A TYPE-CASTING OPERATOR

type-casting-operator → **`is`** type
type-casting-operator → **`as`** type
type-casting-operator → **`as ?`** type
type-casting-operator → **`as !`** type

Primary Expressions

Primary expressions are the most basic kind of expression. They can be used as expressions on their own, and they can be combined with other tokens to make prefix expressions, binary expressions, and postfix expressions.

GRAMMAR OF A PRIMARY EXPRESSION

primary-expression → identifier generic-argument-clause*opt*
primary-expression → literal-expression
primary-expression → self-expression

> *primary-expression* → superclass-expression
> *primary-expression* → closure-expression
> *primary-expression* → parenthesized-expression
> *primary-expression* → tuple-expression
> *primary-expression* → implicit-member-expression
> *primary-expression* → wildcard-expression
> *primary-expression* → key-path-expression
> *primary-expression* → selector-expression
> *primary-expression* → key-path-string-expression

Literal Expression

A *literal expression* consists of either an ordinary literal (such as a string or a number), an array or dictionary literal, a playground literal, or one of the following special literals:

Literal	Type	Value
`#file`	`String`	The name of the file in which it appears.
`#line`	`Int`	The line number on which it appears.
`#column`	`Int`	The column number in which it begins.
`#function`	`String`	The name of the declaration in which it appears.
`#dsohandle`	`UnsafeRawPointer`	The DSO (dynamic shared object) handle in use where it appears.

Inside a function, the value of `#function` is the name of that function, inside a method it is the name of that method, inside a property getter or setter it is the name of that property, inside special members like `init` or `subscript` it is the name of that keyword, and at the top level of a file it is the name of the current module.

When used as the default value of a function or method parameter, the special literal's value is determined when the default value expression is evaluated at the call site.

```
func logFunctionName(string: String = #function) {
    print(string)
}
func myFunction() {
    logFunctionName() // Prints "myFunction()".
}
```

An *array literal* is an ordered collection of values. It has the following form:

```
[value 1, value 2, ...]
```

The last expression in the array can be followed by an optional comma. The value of an array literal has type `[T]`, where `T` is the type of the expressions inside it. If there are expressions of multiple types, `T` is their closest common supertype. Empty array literals are written using an empty pair of square brackets and can be used to create an empty array of a specified type.

```
var emptyArray: [Double] = []
```

A *dictionary literal* is an unordered collection of key-value pairs. It has the following form:

```
[key 1: value 1, key 2: value 2, ...]
```

The last expression in the dictionary can be followed by an optional comma. The value of a dictionary literal has type `[Key: Value]`, where `Key` is the type of its key expressions and `Value` is the type of its value expressions. If there are expressions of multiple types, `Key` and `Value` are the closest common supertype for their respective values. An empty dictionary literal is written as a colon inside a pair of brackets (`[:]`) to distinguish it from an empty array literal. You can use an empty dictionary literal to create an empty dictionary literal of specified key and value types.

```
var emptyDictionary: [String: Double] = [:]
```

A *playground literal* is used by Xcode to create an interactive representation of a color, file, or image within the program editor. Playground literals in plain text outside of Xcode are represented using a special literal syntax.

For information on using playground literals in Xcode, see Add a color, file, or image literal in Xcode Help.

GRAMMAR OF A LITERAL EXPRESSION

literal-expression → literal
literal-expression → array-literal | dictionary-literal | playground-literal
literal-expression → **`#file`** | **`#line`** | **`#column`** | **`#function`** | **`#dsohandle`**

array-literal → **`[`** array-literal-items$_{opt}$ **`]`**
array-literal-items → array-literal-item **`,`**$_{opt}$ | array-literal-item **`,`** array-literal-items
array-literal-item → expression

dictionary-literal → **`[`** dictionary-literal-items **`]`** | **`[`** **`:`** **`]`**
dictionary-literal-items → dictionary-literal-item **`,`**$_{opt}$ | dictionary-literal-item **`,`** dictionary-literal-items
dictionary-literal-item → expression **`:`** expression

playground-literal → **`#colorLiteral`** **`(`** **`red`** **`:`** expression **`,`** **`green`** **`:`** expression **`,`** **`blue`** **`:`** expression **`,`** **`alpha`** **`:`** expression **`)`**
playground-literal → **`#fileLiteral`** **`(`** **`resourceName`** **`:`** expression **`)`**
playground-literal → **`#imageLiteral`** **`(`** **`resourceName`** **`:`** expression **`)`**

Self Expression

The `self` expression is an explicit reference to the current type or instance of the type in which it occurs. It has the following forms:

```
self
self.member name
self[subscript index]
self(initializer arguments)
self.init(initializer arguments)
```

In an initializer, subscript, or instance method, `self` refers to the current instance of the type in which it occurs. In a type method, `self` refers to the current type in which it occurs.

The `self` expression is used to specify scope when accessing members, providing disambiguation when there is another variable of the same name in scope, such as a function parameter. For example:

```
class SomeClass {
    var greeting: String
    init(greeting: String) {
        self.greeting = greeting
    }
}
```

In a mutating method of a value type, you can assign a new instance of that value type to `self`. For example:

```
struct Point {
    var x = 0.0, y = 0.0
    mutating func moveBy(x deltaX: Double, y deltaY: Double)
  {
        self = Point(x: x + deltaX, y: y + deltaY)
    }
}
```

GRAMMAR OF A SELF EXPRESSION

self-expression → **`self`** | self-method-expression | self-subscript-expression | self-initializer-expression

self-method-expression → **self** **.** identifier
self-subscript-expression → **self** **[** function-call-argument-list **]**
self-initializer-expression → **self** **.** **init**

Superclass Expression

A *superclass expression* lets a class interact with its superclass. It has one of the following forms:

```
super.member name
super[subscript index]
super.init(initializer arguments)
```

The first form is used to access a member of the superclass. The second form is used to access the superclass's subscript implementation. The third form is used to access an initializer of the superclass.

Subclasses can use a superclass expression in their implementation of members, subscripting, and initializers to make use of the implementation in their superclass.

GRAMMAR OF A SUPERCLASS EXPRESSION

superclass-expression → superclass-method-expression | superclass-subscript-expression | superclass-initializer-expression

superclass-method-expression → **super** **.** identifier
superclass-subscript-expression → **super** **[** function-call-argument-list **]**
superclass-initializer-expression → **super** **.** **init**

Closure Expression

A *closure expression* creates a closure, also known as a *lambda* or an *anonymous function* in other programming languages. Like a function declaration, a closure contains statements, and it captures constants and variables from its enclosing scope. It has the following form:

```
{ (parameters) -> return type in
    statements
}
```

The *parameters* have the same form as the parameters in a function declaration, as described in Function Declaration.

There are several special forms that allow closures to be written more concisely:

- A closure can omit the types of its parameters, its return type, or both. If you omit the parameter names and both types, omit the `in` keyword before the statements. If the omitted types can't be inferred, a compile-time error is raised.
- A closure may omit names for its parameters. Its parameters are then implicitly named `$` followed by their position: `$0`, `$1`, `$2`, and so on.
- A closure that consists of only a single expression is understood to return the value of that expression. The contents of this expression are also considered when performing type inference on the surrounding expression.

The following closure expressions are equivalent:

```
myFunction { (x: Int, y: Int) -> Int in
    return x + y
}

myFunction { x, y in
    return x + y
}

myFunction { return $0 + $1 }

myFunction { $0 + $1 }
```

For information about passing a closure as an argument to a function, see Function Call Expression.

Closure expressions can be used without being stored in a variable or constant, such as when you immediately use a closure as part of a function call. The closure expressions passed to `myFunction` in code above are examples of this kind of immediate use. As a result, whether a closure expression is escaping or nonescaping depends on the surrounding context of the expression. A closure expression is nonescaping if it is called immediately or passed as a nonescaping function argument. Otherwise, the closure expression is escaping.

For more information about escaping closures, see Escaping Closures.

Capture Lists

By default, a closure expression captures constants and variables from its surrounding scope with strong references to those values. You can use a *capture list* to explicitly control how values are captured in a closure.

A capture list is written as a comma-separated list of expressions surrounded by square brackets, before the list of parameters. If you use a capture list, you must also use the `in` keyword, even if you omit the parameter names, parameter types, and return type.

The entries in the capture list are initialized when the closure is created. For each entry in the capture list, a constant is initialized to the value of the constant or variable that has the same name in the surrounding scope. For example in the code below, `a` is included in the capture list but `b` is not, which gives them different behavior.

```
var a = 0
var b = 0
let closure = { [a] in
    print(a, b)
}

a = 10
b = 10
closure()
// Prints "0 10"
```

There are two different things named a, the variable in the surrounding scope and the constant in the closure's scope, but only one variable named b. The a in the inner scope is initialized with the value of the a in the outer scope when the closure is created, but their values are not connected in any special way. This means that a change to the value of a in the outer scope does not affect the value of a in the inner scope, nor does a change to a inside the closure affect the value of a outside the closure. In contrast, there is only one variable named b—the b in the outer scope—so changes from inside or outside the closure are visible in both places.

This distinction is not visible when the captured variable's type has reference semantics. For example, there are two things named x in the code below, a variable in the outer scope and a constant in the inner scope, but they both refer to the same object because of reference semantics.

```
class SimpleClass {
    var value: Int = 0
}
var x = SimpleClass()
var y = SimpleClass()
let closure = { [x] in
    print(x.value, y.value)
}

x.value = 10
y.value = 10
closure()
// Prints "10 10"
```

If the type of the expression's value is a class, you can mark the expression in a capture list with weak or unowned to capture a weak or unowned reference to the expression's value.

```
myFunction { print(self.title) }                    //
  implicit strong capture
myFunction { [self] in print(self.title) }          //
  explicit strong capture
myFunction { [weak self] in print(self!.title) }    // weak
  capture
myFunction { [unowned self] in print(self.title) }  //
```

```
unowned capture
```

You can also bind an arbitrary expression to a named value in a capture list. The expression is evaluated when the closure is created, and the value is captured with the specified strength. For example:

```
// Weak capture of "self.parent" as "parent"
myFunction { [weak parent = self.parent] in
  print(parent!.title) }
```

For more information and examples of closure expressions, see Closure Expressions. For more information and examples of capture lists, see Resolving Strong Reference Cycles for Closures.

GRAMMAR OF A CLOSURE EXPRESSION

closure-expression → **{** closure-signature$_{opt}$ statements$_{opt}$ **}**

closure-signature → capture-list$_{opt}$ closure-parameter-clause **throws**$_{opt}$ function-result$_{opt}$ **in**
closure-signature → capture-list **in**

closure-parameter-clause → **(** **)** | **(** closure-parameter-list **)** | identifier-list
closure-parameter-list → closure-parameter | closure-parameter **,** closure-parameter-list
closure-parameter → closure-parameter-name type-annotation$_{opt}$
closure-parameter → closure-parameter-name type-annotation **...**
closure-parameter-name → identifier

capture-list → **[** capture-list-items **]**
capture-list-items → capture-list-item | capture-list-item **,** capture-list-items
capture-list-item → capture-specifier$_{opt}$ expression
capture-specifier → **weak** | **unowned** | **unowned(safe)** | **unowned(unsafe)**

Implicit Member Expression

An *implicit member expression* is an abbreviated way to access a member of a type, such as an enumeration case or a type method, in a context where type inference can determine the implied type. It has the following form:

```
.member name
```

For example:

```
var x = MyEnumeration.someValue
x = .anotherValue
```

GRAMMAR OF A IMPLICIT MEMBER EXPRESSION

implicit-member-expression → **.** identifier

Parenthesized Expression

A *parenthesized expression* consists of an expression surrounded by parentheses. You can use parentheses to specify the precedence of operations by explicitly grouping expressions. Grouping parentheses don't change an expression's type—for example, the type of `(1)` is simply `Int`.

GRAMMAR OF A PARENTHESIZED EXPRESSION

parenthesized-expression → **(** expression **)**

Tuple Expression

A *tuple expression* consists of a comma-separated list of expressions surrounded by parentheses. Each expression can have an optional identifier before it, separated by a colon (`:`). It has the following form:

```
(identifier 1: expression 1, identifier 2:
   expression 2, ...)
```

A tuple expression can contain zero expressions, or it can contain two or more expressions. A single expression inside parentheses is a parenthesized expression.

> NOTE
>
> Both an empty tuple expression and an empty tuple type are written `()` in Swift. Because `Void` is a type alias for `()`, you can use it to write an empty tuple type. However, like all type aliases, `Void` is always a type—you can't use it to write an empty tuple expression.

GRAMMAR OF A TUPLE EXPRESSION

tuple-expression → **( )** | **(** tuple-element **,** tuple-element-list **)**
tuple-element-list → tuple-element | tuple-element **,** tuple-element-list
tuple-element → expression | identifier **:** expression

Wildcard Expression

A *wildcard expression* is used to explicitly ignore a value during an assignment. For example, in the following assignment 10 is assigned to x and 20 is ignored:

```
(x, _) = (10, 20)
// x is 10, and 20 is ignored
```

GRAMMAR OF A WILDCARD EXPRESSION

wildcard-expression → **_**

Key-Path Expression

A *key-path expression* refers to a property or subscript of a type. You use key-path expressions in dynamic programming tasks, such as key-value observing. They have the following form:

```
\type name.path
```

The *type name* is the name of a concrete type, including any generic parameters, such as `String`, `[Int]`, or `Set<Int>`.

The *path* consists of property names, subscripts, optional-chaining expressions, and forced unwrapping expressions. Each of these key-path components can be repeated as many times as needed, in any order.

At compile time, a key-path expression is replaced by an instance of the `KeyPath` class.

To access a value using a key path, pass the key path to the `subscript(keyPath:)` subscript, which is available on all types. For example:

```
struct SomeStructure {
    var someValue: Int
}

let s = SomeStructure(someValue: 12)
let pathToProperty = \SomeStructure.someValue

let value = s[keyPath: pathToProperty]
// value is 12
```

The *type name* can be omitted in contexts where type inference can determine the implied type. The following code uses `\.someProperty` instead of `\SomeClass.someProperty`:

```
class SomeClass: NSObject {
    @objc var someProperty: Int
    init(someProperty: Int) {
        self.someProperty = someProperty
    }
}

let c = SomeClass(someProperty: 10)
c.observe(\.someProperty) { object, change in
    // ...
}
```

The *path* can refer to `self` to create the identity key path (`\.self`). The identity key path refers to a whole instance, so you can use it to access and change all of the data stored in a variable in a single step. For example:

```
var compoundValue = (a: 1, b: 2)
// Equivalent to compoundValue = (a: 10, b: 20)
compoundValue[keyPath: \.self] = (a: 10, b: 20)
```

The *path* can contain multiple property names, separated by periods, to refer to a property of a property's value. This code uses the key path expression `\OuterStructure.outer.someValue` to access the `someValue` property of the `OuterStructure` type's `outer` property:

```
struct OuterStructure {
    var outer: SomeStructure
    init(someValue: Int) {
        self.outer = SomeStructure(someValue: someValue)
    }
}

let nested = OuterStructure(someValue: 24)
let nestedKeyPath = \OuterStructure.outer.someValue

let nestedValue = nested[keyPath: nestedKeyPath]
// nestedValue is 24
```

The *path* can include subscripts using brackets, as long as the subscript's parameter type conforms to the `Hashable` protocol. This example uses a subscript in a key path to access the second element of an array:

```
let greetings = ["hello", "hola", "bonjour", "안녕"]
let myGreeting = greetings[keyPath: \[String].[1]]
// myGreeting is 'hola'
```

The value used in a subscript can be a named value or a literal. Values are captured in key paths using value semantics. The following code uses the variable `index` in both a key-path expression and in a closure to access the third element of the `greetings` array. When `index` is modified, the key-path expression still references the third element, while the closure uses the new index.

```
var index = 2
let path = \[String].[index]
let fn: ([String]) -> String = { strings in strings[index] }

print(greetings[keyPath: path])
// Prints "bonjour"
print(fn(greetings))
// Prints "bonjour"

// Setting 'index' to a new value doesn't affect 'path'
index += 1
print(greetings[keyPath: path])
// Prints "bonjour"

// Because 'fn' closes over 'index', it uses the new value
print(fn(greetings))
// Prints "안녕"
```

The *path* can use optional chaining and forced unwrapping. This code uses optional chaining in a key path to access a property of an optional string:

```
let firstGreeting: String? = greetings.first
print(firstGreeting?.count as Any)
// Prints "Optional(5)"

// Do the same thing using a key path.
let count = greetings[keyPath: \[String].first?.count]
print(count as Any)
// Prints "Optional(5)"
```

You can mix and match components of key paths to access values that are deeply nested within a type. The following code accesses different values and properties of a dictionary of arrays by using key-path expressions that combine these components.

```
let interestingNumbers = ["prime": [2, 3, 5, 7, 11, 13, 17],
                          "triangular": [1, 3, 6, 10, 15,
  21, 28],
                          "hexagonal": [1, 6, 15, 28, 45,
  66, 91]]
print(interestingNumbers[keyPath: \[String: [Int]].
  ["prime"]] as Any)
// Prints "Optional([2, 3, 5, 7, 11, 13, 17])"
print(interestingNumbers[keyPath: \[String: [Int]].
  ["prime"]![0]])
// Prints "2"
print(interestingNumbers[keyPath: \[String: [Int]].
  ["hexagonal"]!.count])
// Prints "7"
print(interestingNumbers[keyPath: \[String: [Int]].
  ["hexagonal"]!.count.bitWidth])
// Prints "64"
```

For more information about using key paths in code that interacts with Objective-C APIs, see Using Objective-C Runtime Features in Swift. For information about key-value coding and key-value observing, see Key-Value Coding Programming Guide and Key-Value Observing Programming Guide.

GRAMMAR OF A KEY-PATH EXPRESSION

key-path-expression → \ type$_{opt}$ **.** key-path-components
key-path-components → key-path-component | key-path-component **.** key-path-components
key-path-component → identifier key-path-postfixes$_{opt}$ | key-path-postfixes

key-path-postfixes → key-path-postfix key-path-postfixes$_{opt}$
key-path-postfix → **?** | **!** | **self** | **[** function-call-argument-list **]**

Selector Expression

A selector expression lets you access the selector used to refer to a method or to a property's getter or setter in Objective-C. It has the following form:

```
#selector( method name )
#selector(getter: property name )
#selector(setter: property name )
```

The *method name* and *property name* must be a reference to a method or a property that is available in the Objective-C runtime. The value of a selector expression is an instance of the `Selector` type. For example:

```
class SomeClass: NSObject {
    @objc let property: String
    @objc(doSomethingWithInt:)
    func doSomething(_ x: Int) {}

    init(property: String) {
        self.property = property
    }
}
let selectorForMethod = #selector(SomeClass.doSomething(_:))
let selectorForPropertyGetter = #selector(getter:
  SomeClass.property)
```

When creating a selector for a property's getter, the *property name* can be a reference to a variable or constant property. In contrast, when creating a selector for a property's setter, the *property name* must be a reference to a variable property only.

The *method name* can contain parentheses for grouping, as well the `as` operator to disambiguate between methods that share a name but have different type signatures. For example:

```
extension SomeClass {
    @objc(doSomethingWithString:)
    func doSomething(_ x: String) { }
}
let anotherSelector = #selector(SomeClass.doSomething(_:) as
  (SomeClass) -> (String) -> Void)
```

Because a selector is created at compile time, not at runtime, the compiler can check that a method or property exists and that they're exposed to the Objective-C runtime.

> NOTE
>
> Although the *method name* and the *property name* are expressions, they're never evaluated.

For more information about using selectors in Swift code that interacts with Objective-C APIs, see Using Objective-C Runtime Features in Swift.

GRAMMAR OF A SELECTOR EXPRESSION

selector-expression → **#selector** **(** expression **)**
selector-expression → **#selector** **(** **getter:** expression **)**
selector-expression → **#selector** **(** **setter:** expression **)**

Key-Path String Expression

A key-path string expression lets you access the string used to refer to a property in Objective-C, for use in key-value coding and key-value observing APIs. It has the following form:

```
#keyPath( property name )
```

The *property name* must be a reference to a property that is available in the Objective-C runtime. At compile time, the key-path string expression is replaced by a string literal. For example:

```
class SomeClass: NSObject {
    @objc var someProperty: Int
    init(someProperty: Int) {
        self.someProperty = someProperty
    }
}

let c = SomeClass(someProperty: 12)
let keyPath = #keyPath(SomeClass.someProperty)

if let value = c.value(forKey: keyPath) {
    print(value)
}
// Prints "12"
```

When you use a key-path string expression within a class, you can refer to a property of that class by writing just the property name, without the class name.

```
extension SomeClass {
    func getSomeKeyPath() -> String {
        return #keyPath(someProperty)
    }
}
print(keyPath == c.getSomeKeyPath())
// Prints "true"
```

Because the key path string is created at compile time, not at runtime, the compiler can check that the property exists and that the property is exposed to the Objective-C runtime.

For more information about using key paths in Swift code that interacts with Objective-C APIs, see Using Objective-C Runtime Features in Swift. For information about key-value coding and key-value observing, see Key-Value Coding Programming Guide and Key-Value Observing Programming Guide.

> NOTE
>
> Although the *property name* is an expression, it is never evaluated.

GRAMMAR OF A KEY-PATH STRING EXPRESSION

key-path-string-expression → **`#keyPath`** **`(`** expression **`)`**

Postfix Expressions

Postfix expressions are formed by applying a postfix operator or other postfix syntax to an expression. Syntactically, every primary expression is also a postfix expression.

For information about the behavior of these operators, see Basic Operators and Advanced Operators.

For information about the operators provided by the Swift standard library, see Operator Declarations.

GRAMMAR OF A POSTFIX EXPRESSION

postfix-expression → primary-expression
postfix-expression → postfix-expression postfix-operator
postfix-expression → function-call-expression
postfix-expression → initializer-expression
postfix-expression → explicit-member-expression
postfix-expression → postfix-self-expression
postfix-expression → subscript-expression
postfix-expression → forced-value-expression
postfix-expression → optional-chaining-expression

Function Call Expression

A *function call expression* consists of a function name followed by a comma-separated list of the function's arguments in parentheses. Function call expressions have the following form:

```
function name(argument value 1, argument value 2)
```

The *function name* can be any expression whose value is of a function type.

If the function definition includes names for its parameters, the function call must include names before its argument values separated by a colon (`:`). This kind of function call expression has the following form:

```
function name(argument name 1: argument value 1,
   argument name 2: argument value 2)
```

A function call expression can include a trailing closure in the form of a closure expression immediately after the closing parenthesis. The trailing closure is understood as an argument to the function, added after the last parenthesized argument. The following function calls are equivalent:

```
// someFunction takes an integer and a closure as its
  arguments
someFunction(x: x, f: {$0 == 13})
someFunction(x: x) {$0 == 13}
```

If the trailing closure is the function's only argument, the parentheses can be omitted.

```
// someMethod takes a closure as its only argument
myData.someMethod() {$0 == 13}
myData.someMethod {$0 == 13}
```

GRAMMAR OF A FUNCTION CALL EXPRESSION

function-call-expression → postfix-expression function-call-argument-clause

function-call-expression → postfix-expression function-call-argument-clause$_{opt}$ trailing-closure

function-call-argument-clause → **(** **)** | **(** function-call-argument-list **)**

function-call-argument-list → function-call-argument | function-call-argument **,** function-call-argument-list

> *function-call-argument* → expression | identifier **:** expression
> *function-call-argument* → operator | identifier **:** operator
>
> *trailing-closure* → closure-expression

Initializer Expression

An *initializer expression* provides access to a type's initializer. It has the following form:

```
expression.init(initializer arguments)
```

You use the initializer expression in a function call expression to initialize a new instance of a type. You also use an initializer expression to delegate to the initializer of a superclass.

```swift
class SomeSubClass: SomeSuperClass {
    override init() {
        // subclass initialization goes here
        super.init()
    }
}
```

Like a function, an initializer can be used as a value. For example:

```swift
// Type annotation is required because String has multiple
  initializers.
let initializer: (Int) -> String = String.init
let oneTwoThree = [1, 2, 3].map(initializer).reduce("", +)
print(oneTwoThree)
// Prints "123"
```

If you specify a type by name, you can access the type's initializer without using an initializer expression. In all other cases, you must use an initializer expression.

```swift
let s1 = SomeType.init(data: 3)  // Valid
let s2 = SomeType(data: 1)       // Also valid

let s3 = type(of: someValue).init(data: 7)  // Valid
let s4 = type(of: someValue)(data: 5)       // Error
```

GRAMMAR OF AN INITIALIZER EXPRESSION

initializer-expression → postfix-expression **.** **init**
initializer-expression → postfix-expression **.** **init** **(** argument-names **)**

Explicit Member Expression

An *explicit member expression* allows access to the members of a named type, a tuple, or a module. It consists of a period (`.`) between the item and the identifier of its member.

```
expression.member name
```

The members of a named type are named as part of the type's declaration or extension. For example:

```
class SomeClass {
    var someProperty = 42
}
let c = SomeClass()
let y = c.someProperty  // Member access
```

The members of a tuple are implicitly named using integers in the order they appear, starting from zero. For example:

```
var t = (10, 20, 30)
t.0 = t.1
// Now t is (20, 20, 30)
```

The members of a module access the top-level declarations of that module.

Types declared with the `dynamicMemberLookup` attribute include members that are looked up at runtime, as described in Attributes.

To distinguish between methods or initializers whose names differ only by the names of their arguments, include the argument names in parentheses, with each argument name followed by a colon (`:`). Write an underscore (`_`) for an argument with no name. To distinguish between overloaded methods, use a type annotation. For example:

```
class SomeClass {
    func someMethod(x: Int, y: Int) {}
    func someMethod(x: Int, z: Int) {}
    func overloadedMethod(x: Int, y: Int) {}
    func overloadedMethod(x: Int, y: Bool) {}
}
let instance = SomeClass()

let a = instance.someMethod              // Ambiguous
let b = instance.someMethod(x:y:)        // Unambiguous

let d = instance.overloadedMethod        // Ambiguous
let d = instance.overloadedMethod(x:y:)  // Still ambiguous
let d: (Int, Bool) -> Void  =
  instance.overloadedMethod(x:y:)  // Unambiguous
```

If a period appears at the beginning of a line, it is understood as part of an explicit member expression, not as an implicit member expression. For example, the following listing shows chained method calls split over several lines:

```
let x = [10, 3, 20, 15, 4]
    .sorted()
    .filter { $0 > 5 }
    .map { $0 * 100 }
```

GRAMMAR OF AN EXPLICIT MEMBER EXPRESSION

explicit-member-expression → postfix-expression **.** decimal-digits
explicit-member-expression → postfix-expression **.** identifier generic-argument-clause$_{opt}$
explicit-member-expression → postfix-expression **.** identifier **(** argument-names **)**

argument-names → argument-name argument-names$_{opt}$
argument-name → identifier **:**

Postfix Self Expression

A postfix `self` expression consists of an expression or the name of a type, immediately followed by `.self`. It has the following forms:

```
expression.self
type.self
```

The first form evaluates to the value of the *expression*. For example, `x.self` evaluates to `x`.

The second form evaluates to the value of the *type*. Use this form to access a type as a value. For example, because `SomeClass.self` evaluates to the `SomeClass` type itself, you can pass it to a function or method that accepts a type-level argument.

> GRAMMAR OF A POSTFIX SELF EXPRESSION
>
> *postfix-self-expression* → postfix-expression **.** **self**

Subscript Expression

A *subscript expression* provides subscript access using the getter and setter of the corresponding subscript declaration. It has the following form:

```
expression[index expressions]
```

To evaluate the value of a subscript expression, the subscript getter for the *expression*'s type is called with the *index expressions* passed as the subscript parameters. To set its value, the subscript setter is called in the same way.

For information about subscript declarations, see Protocol Subscript Declaration.

> GRAMMAR OF A SUBSCRIPT EXPRESSION
>
> *subscript-expression* → postfix-expression **[** function-call-argument-list **]**

Forced-Value Expression

A *forced-value expression* unwraps an optional value that you are certain is not `nil`. It has the following form:

```
expression!
```

If the value of the *expression* is not `nil`, the optional value is unwrapped and returned with the corresponding non-optional type. Otherwise, a runtime error is raised.

The unwrapped value of a forced-value expression can be modified, either by mutating the value itself, or by assigning to one of the value's members. For example:

```swift
var x: Int? = 0
x! += 1
// x is now 1

var someDictionary = ["a": [1, 2, 3], "b": [10, 20]]
someDictionary["a"]![0] = 100
// someDictionary is now ["a": [100, 2, 3], "b": [10, 20]]
```

GRAMMAR OF A FORCED-VALUE EXPRESSION

forced-value-expression → postfix-expression **!**

Optional-Chaining Expression

An *optional-chaining expression* provides a simplified syntax for using optional values in postfix expressions. It has the following form:

```
expression?
```

The postfix `?` operator makes an optional-chaining expression from an expression without changing the expression's value.

Optional-chaining expressions must appear within a postfix expression, and they cause the postfix expression to be evaluated in a special way. If the value of the optional-chaining expression is `nil`, all of the other operations in the postfix expression are ignored and the entire postfix expression evaluates to `nil`. If the value of the optional-chaining expression is not `nil`, the value of the optional-chaining expression is unwrapped and used to evaluate the rest of the postfix expression. In either case, the value of the postfix expression is still of an optional type.

If a postfix expression that contains an optional-chaining expression is nested inside other postfix expressions, only the outermost expression returns an optional type. In the example below, when `c` is not `nil`, its value is unwrapped and used to evaluate `.property`, the value of which is used to evaluate `.performAction()`. The entire expression `c?.property.performAction()` has a value of an optional type.

```swift
var c: SomeClass?
var result: Bool? = c?.property.performAction()
```

The following example shows the behavior of the example above without using optional chaining.

```swift
var result: Bool?
if let unwrappedC = c {
    result = unwrappedC.property.performAction()
}
```

The unwrapped value of an optional-chaining expression can be modified, either by mutating the value itself, or by assigning to one of the value's members. If the value of the optional-chaining expression is `nil`, the expression on the right-hand side of the assignment operator is not evaluated. For example:

```swift
func someFunctionWithSideEffects() -> Int {
    return 42  // No actual side effects.
}
var someDictionary = ["a": [1, 2, 3], "b": [10, 20]]

someDictionary["not here"]?[0] =
  someFunctionWithSideEffects()
// someFunctionWithSideEffects is not evaluated
// someDictionary is still ["a": [1, 2, 3], "b": [10, 20]]

someDictionary["a"]?[0] = someFunctionWithSideEffects()
// someFunctionWithSideEffects is evaluated and returns 42
// someDictionary is now ["a": [42, 2, 3], "b": [10, 20]]
```

GRAMMAR OF AN OPTIONAL-CHAINING EXPRESSION

optional-chaining-expression → postfix-expression **?**

< Types Statements >