Swift

Patterns

A *pattern* represents the structure of a single value or a composite value. For example, the structure of a tuple `(1, 2)` is a comma-separated list of two elements. Because patterns represent the structure of a value rather than any one particular value, you can match them with a variety of values. For instance, the pattern `(x, y)` matches the tuple `(1, 2)` and any other two-element tuple. In addition to matching a pattern with a value, you can extract part or all of a composite value and bind each part to a constant or variable name.

In Swift, there are two basic kinds of patterns: those that successfully match any kind of value, and those that may fail to match a specified value at runtime.

The first kind of pattern is used for destructuring values in simple variable, constant, and optional bindings. These include wildcard patterns, identifier patterns, and any value binding or tuple patterns containing them. You can specify a type annotation for these patterns to constrain them to match only values of a certain type.

The second kind of pattern is used for full pattern matching, where the values you're trying to match against may not be there at runtime. These include enumeration case patterns, optional patterns, expression patterns, and type-casting patterns. You use these patterns in a case label of a `switch` statement, a `catch` clause of a `do` statement, or in the case condition of an `if`, `while`, `guard`, or `for-in` statement.

GRAMMAR OF A PATTERN

pattern → wildcard-pattern type-annotation$_{opt}$
pattern → identifier-pattern type-annotation$_{opt}$
pattern → value-binding-pattern
pattern → tuple-pattern type-annotation$_{opt}$
pattern → enum-case-pattern

pattern → optional-pattern
pattern → type-casting-pattern
pattern → expression-pattern

Wildcard Pattern

A *wildcard pattern* matches and ignores any value and consists of an underscore (_). Use a wildcard pattern when you don't care about the values being matched against. For example, the following code iterates through the closed range `1...3`, ignoring the current value of the range on each iteration of the loop:

```
for _ in 1...3 {
    // Do something three times.
}
```

GRAMMAR OF A WILDCARD PATTERN

wildcard-pattern → **_**

Identifier Pattern

An *identifier pattern* matches any value and binds the matched value to a variable or constant name. For example, in the following constant declaration, `someValue` is an identifier pattern that matches the value `42` of type `Int`:

```
let someValue = 42
```

When the match succeeds, the value `42` is bound (assigned) to the constant name `someValue`.

When the pattern on the left-hand side of a variable or constant declaration is an identifier pattern, the identifier pattern is implicitly a subpattern of a value-binding pattern.

GRAMMAR OF AN IDENTIFIER PATTERN

identifier-pattern → identifier

Value-Binding Pattern

A *value-binding pattern* binds matched values to variable or constant names. Value-binding patterns that bind a matched value to the name of a constant begin with the `let` keyword; those that bind to the name of variable begin with the `var` keyword.

Identifiers patterns within a value-binding pattern bind new named variables or constants to their matching values. For example, you can decompose the elements of a tuple and bind the value of each element to a corresponding identifier pattern.

```
let point = (3, 2)
switch point {
    // Bind x and y to the elements of point.
case let (x, y):
    print("The point is at (\(x), \(y)).")
}
// Prints "The point is at (3, 2)."
```

In the example above, `let` distributes to each identifier pattern in the tuple pattern `(x, y)`. Because of this behavior, the `switch` cases `case let (x, y):` and `case (let x, let y):` match the same values.

GRAMMAR OF A VALUE-BINDING PATTERN

value-binding-pattern → **var** pattern | **let** pattern

Tuple Pattern

A *tuple pattern* is a comma-separated list of zero or more patterns, enclosed in parentheses. Tuple patterns match values of corresponding tuple types.

You can constrain a tuple pattern to match certain kinds of tuple types by using type annotations. For example, the tuple pattern `(x, y): (Int, Int)` in the constant declaration `let (x, y): (Int, Int) = (1, 2)` matches only tuple types in which both elements are of type `Int`.

When a tuple pattern is used as the pattern in a `for-in` statement or in a variable or constant declaration, it can contain only wildcard patterns, identifier patterns, optional patterns, or other tuple patterns that contain those. For example, the following code isn't valid because the element `0` in the tuple pattern `(x, 0)` is an expression pattern:

```
let points = [(0, 0), (1, 0), (1, 1), (2, 0), (2, 1)]
// This code isn't valid.
for (x, 0) in points {
    /* ... */
}
```

The parentheses around a tuple pattern that contains a single element have no effect. The pattern matches values of that single element's type. For example, the following are equivalent:

```
let a = 2        // a: Int = 2
let (a) = 2      // a: Int = 2
let (a): Int = 2 // a: Int = 2
```

GRAMMAR OF A TUPLE PATTERN

tuple-pattern → **(** tuple-pattern-element-list$_{opt}$ **)**

tuple-pattern-element-list → tuple-pattern-element | tuple-pattern-element **,** tuple-pattern-element-list

tuple-pattern-element → pattern | identifier **:** pattern

Enumeration Case Pattern

An *enumeration case pattern* matches a case of an existing enumeration type. Enumeration case patterns appear in `switch` statement case labels and in the case conditions of `if`, `while`, `guard`, and `for-in` statements.

If the enumeration case you're trying to match has any associated values, the corresponding enumeration case pattern must specify a tuple pattern that contains one element for each associated value. For an example that uses a `switch` statement to match enumeration cases containing associated values, see Associated Values.

GRAMMAR OF AN ENUMERATION CASE PATTERN

enum-case-pattern → type-identifier$_{opt}$ **.** enum-case-name tuple-pattern$_{opt}$

Optional Pattern

An *optional pattern* matches values wrapped in a `some(Wrapped)` case of an `Optional<Wrapped>` enumeration. Optional patterns consist of an identifier pattern followed immediately by a question mark and appear in the same places as enumeration case patterns.

Because optional patterns are syntactic sugar for `Optional` enumeration case patterns, the following are equivalent:

```swift
let someOptional: Int? = 42
// Match using an enumeration case pattern.
if case .some(let x) = someOptional {
    print(x)
}

// Match using an optional pattern.
if case let x? = someOptional {
    print(x)
}
```

The optional pattern provides a convenient way to iterate over an array of optional values in a `for`-`in` statement, executing the body of the loop only for non-`nil` elements.

```swift
let arrayOfOptionalInts: [Int?] = [nil, 2, 3, nil, 5]
// Match only non-nil values.
for case let number? in arrayOfOptionalInts {
    print("Found a \(number)")
}
// Found a 2
// Found a 3
// Found a 5
```

GRAMMAR OF AN OPTIONAL PATTERN

optional-pattern → identifier-pattern **?**

Type-Casting Patterns

There are two type-casting patterns, the `is` pattern and the `as` pattern. The `is` pattern appears only in `switch` statement case labels. The `is` and `as` patterns have the following form:

```
is type
pattern as type
```

The `is` pattern matches a value if the type of that value at runtime is the same as the type specified in the right-hand side of the `is` pattern—or a subclass of that type. The `is` pattern behaves like the `is` operator in that they both perform a type cast but discard the returned type.

The `as` pattern matches a value if the type of that value at runtime is the same as the type specified in the right-hand side of the `as` pattern—or a subclass of that type. If the match succeeds, the type of the matched value is cast to the *pattern* specified in the right-hand side of the `as` pattern.

For an example that uses a `switch` statement to match values with `is` and `as` patterns, see Type Casting for Any and AnyObject.

GRAMMAR OF A TYPE CASTING PATTERN

type-casting-pattern → is-pattern | as-pattern
is-pattern → **is** type
as-pattern → pattern **as** type

Expression Pattern

An *expression pattern* represents the value of an expression. Expression patterns appear only in `switch` statement case labels.

The expression represented by the expression pattern is compared with the value of an input expression using the Swift standard library ~= operator. The matches succeeds if the ~= operator returns `true`. By default, the ~= operator compares two values of the same type using the == operator. It can also match a value with a range of values, by checking whether the value is contained within the range, as the following example shows.

```
let point = (1, 2)
switch point {
case (0, 0):
    print("(0, 0) is at the origin.")
case (-2...2, -2...2):
    print("(\(point.0), \(point.1)) is near the origin.")
default:
    print("The point is at (\(point.0), \(point.1)).")
}
// Prints "(1, 2) is near the origin."
```

You can overload the ~= operator to provide custom expression matching behavior. For example, you can rewrite the above example to compare the `point` expression with a string representations of points.

```
// Overload the ~= operator to match a string with an
  integer.
func ~= (pattern: String, value: Int) -> Bool {
    return pattern == "\(value)"
}
switch point {
case ("0", "0"):
    print("(0, 0) is at the origin.")
default:
    print("The point is at (\(point.0), \(point.1)).")
}
// Prints "The point is at (1, 2)."
```

GRAMMAR OF AN EXPRESSION PATTERN

expression-pattern → expression

< Attributes Generic Parameters and Arguments >

BETA SOFTWARE

This documentation contains preliminary information about an API or technology in development. This information is subject to change, and software implemented according to this documentation should be tested with final operating system software.

Learn more about using Apple's beta software