Argument Matching for Trailing Closures

Where trailing closures are used to pass one or more arguments to a function, the argument label for the first trailing closure is always omitted:

```
func animate(
  withDuration duration: Double,
  animations: () -> Void,
  completion: (() -> Void)? = nil
) { /* ... */ }

animate(withDuration: 0.3) /* `animations:` is unwritten. */ {
  // Animate something.
} completion: {
  // Completion handler.
}
```

Sometimes, an unlabeled trailing closure argument can be matched to more than one function parameter. Before Swift 5.3, the compiler would use a **backward scanning rule** to match the unlabeled trailing closure, scanning backwards from the end of the parameter list until finding a parameter that can accept a closure argument (a function type, unconstrained generic type, `Any`, etc.):

```
animate(withDuration: 0.3) {
  // Animate something?
  // The compiler matches this to the `completion` parameter.
}
// error: missing argument for parameter 'animations' in call
```

We encounter a compiler error in this example because the backward scanning rule matches the trailing closure to the `completion` parameter instead of the `animations` parameter, even though `completion` has a default value while `animations` does not.

Swift 5.3 introduces a new **forward scanning rule**, which matches trailing closures to function parameters from left to right (after matching non-trailing arguments). This leads to more predictable and easy-to-understand behavior in many situations. With the new rule, the unlabeled closure in the example above is matched to the `animations` parameter, just as most users would expect:

```
animate(withDuration: 0.3) {
  // Animate something.
} // `completion` has the default value `nil`.
```

When scanning forwards to match an unlabeled trailing closure argument, the compiler will skip any parameter that does not **structurally resemble** a function type and also apply a **heuristic** to skip parameters that do not require an argument in favor of a subsequent parameter that does (see below). These rules make possible the ergonomic use of a modified version of the API given in

the example above, where `withDuration` has a default value:

```
func animate(
  withDuration duration: Double = 1.0,
  animations: () -> Void,
  completion: (() -> Void)? = nil
) { /* ... */ }

animate {
  // Animate something.
  //
  // The closure is not matched to `withDuration` but to `animations` because
  // the first parameter doesn't structurally resemble a function type.
  //
  // If, in place of `withDuration`, there is a parameter with a default value
  // that does structurally resemble a function type, the closure would still be
  // matched to `animations` because it requires an argument while the first
  // parameter does not.
}
```

For source compatibility in Swift 5, the compiler will attempt to apply *both* the new forward scanning rule and the old backward scanning rule when it encounters a function call with a single trailing closure. If the forward and backward scans produce *different valid* matches of arguments to parameters, the compiler will prefer the result of the backward scanning rule and produce a warning. To silence this warning, rewrite the function call to label the argument explicitly without using trailing closure syntax.

Structural Resemblance to a Function Type

A parameter structurally resembles a function type if both of the following are true:

- the parameter is not `inout`, and
- the **adjusted type** of the parameter is a function type.

The adjusted type of the parameter is the parameter's type as it appears in the function declaration, looking through any type aliases, and performing three additional adjustments:

1. If the parameter is an `@autoclosure`, use the result type of the parameter's declared (function) type before performing the second adjustment.
2. If the parameter is variadic, look at the base element type.
3. Remove all outer "optional" types.

Heuristic for Skipping Parameters

To maintain source compatibility, the forward scanning rule applies an additional heuristic when matching trailing closure arguments. If:

- the parameter that would match an unlabeled trailing closure argument according to the forward scanning rule does not require an argument (because it is variadic or has a default argument), *and*
- there are parameters *following* that parameter that *do* require an argument, which appear before the first parameter whose label matches that of the *next* trailing closure (if any),

then the compiler does not match the unlabeled trailing closure to that parameter. Instead, it examines the next parameter to see if that should be matched to the unlabeled trailing closure, as in the following example:

```
func showAlert(
  message: String,
  onPresentation: (() ->  Void)? = nil,
  onDismissal: () -> Void
) { /* ... */ }

// `onPresentation` does not require an argument, but `onDismissal` does, and
// there is no subsequent trailing closure labeled `onDismissal`.
// Therefore, the unlabeled trailing closure is matched to `onDismissal`.
showAlert(message: "Hello, World!") {
  // On dismissal action.
}

// Although `onPresentation` does not require an argument, there are no
// subsequent parameters that require an argument before the parameter whose
// label matches the next trailing closure (`onDismissal`).
// Therefore, the unlabeled trailing closure is matched to `onPresentation`.
showAlert(message: "Hello, World!") {
  // On presentation action.
} onDismissal: {
  // On dismissal action.
}
```

To learn more about argument matching for trailing closures, see Swift Evolution Proposal SE-0286.