orphan:

> **Warning**
>
> This proposal was rejected. We explored a number of alternate syntaxes for method names that both allowed describing arguments and were compatible with Objective-C selectors.

Summary

We make the following incremental language changes:

- We make it so that selector-style declaration syntax declares a **selector name** for a method. `func foo(_ x:T) bar(y:U)` declares a method named with a new selector reference syntax, `self.foo:bar:`. This method cannot be referenced as `foo`; it can only be referenced as `self.foo:bar:`, or with keyword application syntax, `self.foo(x, bar: y)`.
- We change keywords in parens, as in `(a: x, b: y)` to be a feature of apply syntax, instead of a feature of tuple literals. Name lookup changes to resolve keywords according to the function declaration rather than to the context tuple type. For tuple-style declarations `func foo(_ a: Int, b: Int)`, keywords are optional and can be reordered. For selector-style declarations `func foo(_ a: Int) bar(b: Int)`, the keywords are required and must appear in order.
- With keyword arguments no longer reliant on tuple types, we simplify the tuple type system by removing keywords from tuple types.

Keyword Apply Syntax

```
expr-apply ::= expr-postfix '(' (kw-arg (',' kw-arg)*)? ')'
kw-arg ::= (identifier ':')? expr

// Examples
foo()
foo(1, 2, 3)
foo(1, bar: 2, bas: 3)
foo!(1, bar: 2, bas: 3)
a.foo?(1, bar: 2, bas: 3)
```

Keyword syntax becomes a feature of apply expressions. When a named function or method is applied, a declaration is looked up using the name and any keyword arguments that are provided. An arbitrary expression that produces a function result may be applied, but cannot be used with keyword arguments.

Named Application Lookup

A named application is matched to a declaration using both the base name, which appears to the left of the open parens, and the keyword arguments that are provided. The matching rules are different for "tuple-style" and "keyword-style" declarations:

Finding the base name

The callee of an apply is an arbitrary expression. If the expression is a standalone declaration reference or a member reference, then the apply is a **named application**, and the name is used as the **base name** during function name lookup, as described below. Certain expression productions are looked through when looking for a base name:

- The `Optional` postfix operators `!` and `?`. `foo.bar!()` and `bas?()` are named applications, as is `foo?!?()`.
- Parens. `(foo.bar)()` is a named application.

These productions are looked through to arbitrary depth, so `((foo!?)?!)()` is also a named application.

Unnamed applications cannot use keyword arguments.

Tuple-Style Declarations

A tuple-style declaration matches a named application if:

- The base name matches the name of the declaration:

  ```
  func foo(_ x: Int, y: Int) {}
  foo(1, 2) // matches
  bar(1, 2) // doesn't match
  ```

- Positional arguments, that is, arguments without keywords, must be convertible to the type of the corresponding positional parameter of the declaration:

  ```
  func foo(_ x: Int, y: Int) {}
  foo(1, 2) // matches
  foo("one", 2) // doesn't match
  foo(1, "two") // doesn't match
  foo("one", "two") // doesn't match
  ```

- Keyword names, if present, must match the declared name of a parameter in the declaration, and the corresponding argument

must be convertible to the type of the parameter in the declaration. The same keyword name may not be used multiple times, and the same argument cannot be provided positionally and by keyword. All keyword arguments must follow all positional arguments:

```
func foo(_ x: Int, y: String, z: UnicodeScalar) {}
foo(1, "two", '3')          // matches
foo(1, "two", z: '3')       // matches
foo(1, y: "two", '3')       // invalid, positional arg after keyword arg
foo(1, z: '3', y: "two")    // matches
foo(z: '3', x: 1, y: "two") // matches
foo(z: '3', q: 1, y: "two") // doesn't match; no keyword 'q'
foo(1, "two", '3', x: 4)    // doesn't match; 'x' already given positionally
foo(1, "two", z: '3', z: '4') // doesn't match; multiple 'z'
```

As an exception, a trailing closure argument always positionally matches the last declared parameter, and can appear after keyword arguments:

```
func foo(_ x: Int, y: String, f: () -> ()
foo(y: "two", x: 1) { } // matches
```

- If the final declared keyword parameter takes a variadic argument, the keyword in the argument list may be followed by multiple non-keyworded arguments. All arguments up to either the next keyword or the end of the argument list become that keyword's argument:

```
func foo(_ x: Int, y: String, z: UnicodeScalar...) {}
foo(1, "two", '3', '4', '5')       // matches, z = ['3', '4', '5']
foo(1, "two", z: '3', '4', '5')    // same
foo(1, z: '3', '4', '5', y: "two") // same
```

- If the base name is the name of a non-function definition of function type, the input types match the input type of the referenced function value, and there are no keyword arguments:

```
var foo: (Int, Int) -> ()
foo(1, 2) // matches
foo(x: 1, y: 2) // doesn't match
```

Selector-Style Declarations

A selector-style declaration matches a named application if:

- The expression must provide keywords for all of its arguments but the first. It must *not* provide a keyword for the first argument:

```
func foo(_ x: Int) bar(y: String) bas(z: UnicodeScalar) {}
foo(1, "two", '3')              // doesn't match; no keywords
foo(x: 1, bar: "two", bas: '3') // doesn't match; first keyword provided
foo(1, bar: "two", bas: '3')    // matches
```

- The base name of the apply expression must match the first declared selector piece. The subsequent argument keyword names must match the remaining selector pieces in order. The same keyword name may be used multiple times, to refer to selector pieces with the same name. The argument values must be convertible to the declared types of each selector piece's parameter:

```
func foo(_ x: Int) bar(y: String) bas(z: UnicodeScalar) {}
foo(1, bar: "two", bas: '3') // matches
foo(1, bas: '3', bar: "two") // doesn't match; wrong selector piece order
foo(1, bar: '2', bas: "three") // doesn't match; wrong types

func foo(_ x: Int) foo(y: String) foo(z: UnicodeScalar) {}
foo(1, foo: "two", foo: '3') // matches
```

- If the final selector piece declares a variadic parameter, then the keyword in the call expression may be followed by multiple arguments. All arguments up to the end of the argument list become the keyword parameter's value. (Because of strict keyword ordering, additional keywords may not follow.) For example:

```
func foo(_ x: Int) bar(y: String...) {}

foo(1, bar: "two", "three", "four") // matches, y = ["two", "three", "four"]
```

- If the final selector piece declares a function parameter, then the function can be called using trailing closure syntax omitting the keyword. The keyword is still required when trailing closure syntax is not used. For example:

```
func foo(_ x: Int) withBlock(f: () -> ())

foo(1, withBlock: { }) // matches
foo(1, { }) // doesn't match
foo(1) { } // matches
```

Trailing closure syntax can introduce ambiguities when selector-style functions differ only in their final closure selector piece:

```
func foo(_ x: Int) onCompletion(f: () -> ())
func foo(_ x: Int) onError(f: () -> ())

foo(1) { } // error: ambiguous
```

Duplicate Definitions

Tuple-Style Declarations

Keyword names are part of a tuple-style declaration, but they are not part of the declaration's name, they are not part of the declaration's type, and they are not part of the declaration's ABI. Two tuple-style declarations that differ only in keyword names are considered duplicates:

```
// Error: Duplicate definition of foo(Int, Int) -> ()
func foo(_ a: Int, b: Int) {}
func foo(_ x: Int, y: Int) {}
```

Selector-Style Declarations

The name of a selector-style declaration comprises all of its selector pieces in declaration order. Selector-style declarations can be overloaded by selector name, by selector order, and by type:

```
// OK, no duplicates
func foo(_ x: Int) bar(y: Int) bas(z: Int)
func foo(_ x: Int) bar(y: Int) zim(z: Int)
func foo(_ x: Int) bas(y: Int) bar(z: Int)
func foo(_ x: Int) bar(y: Int) bas(z: Float)
```

Tuple- and selector-style declarations are not considered duplicates, even if they can match the same keywords with the same types:

```
// OK, not duplicates
func foo(_ x: Int, bar: Int)
func foo(_ x: Int) bar(x: Int)
```

Unapplied Name Lookup

An unapplied declaration reference `identifier` or member reference `obj.identifier` finds any tuple-style declaration whose name matches the referenced name. It never finds selector-style declarations:

```
func foo(_ a: Int, b: Int) {}
func foo(_ a: Int) bar(b: Int) {}

var f = foo // Finds foo(Int, Int) -> (), not foo:bar:
```

Selector Name Lookup

```
expr-selector-member-ref ::= expr-postfix '.' identifier ':' (identifier ':')+
```

Unapplied selector-style declarations can be referenced as a member of their enclosing context using selector member reference expressions. The name must consist of at least two selector pieces, each followed by a colon. (A single identifier followed by a colon, such as `foo.bar:`, is parsed as a normal member reference `foo.bar` followed by a colon.) A selector member reference expression finds any selector-style declarations whose selector pieces match the named selector pieces in order:

```
class C {
  func foo(_ a: Int) bar(b: Int) bas(c: Int)
  func foo(_ a: Int) bas(b: Int) bar(c: Int)

  func foo(_ a: Int, bar: Int, bas: Int)
}

var c: C

c.foo:bar:bas: // Finds c.foo:bar:bas: (not c.foo or c.foo:bas:bar:)
c.foo:bas:bar: // Finds c.foo:bas:bar:
c.foo          // Finds c.foo
```

QoI Issues

Under this proposal, keyword resolution relies on being able to find a named function declaration. This means that keywords cannot be used with arbitrary expressions of function type. We however still need to parse keywords in nameless applications for recovery. There are also functional operators like `!` and `?` that we need to forward keyword arguments through. Are there others? What about parens? `(foo)(bar: x)` should probably work.

This proposal also prevents a single-element name from being referenced with selector syntax as `foo.bar:`. For QoI, we should

recognize attempts to reference a member in this way, such as `if var f = foo.bar: {}` or `[foo.bar:: bas]`, and fixit away the trailing colon.