String Processing For Swift 4

- Authors: Dave Abrahams, Ben Cohen

The goal of re-evaluating Strings for Swift 4 has been fairly ill-defined thus far, with just this short blurb in the list of goals:

> ***String re-evaluation***: *String is one of the most important fundamental types in the language. The standard library leads have numerous ideas of how to improve the programming model for it, without jeopardizing the goals of providing a unicode-correct-by-default model. Our goal is to be better at string processing than Perl!*

For Swift 4 and beyond we want to improve three dimensions of text processing:

1. Ergonomics
2. Correctness
3. Performance

This document is meant to both provide a sense of the long-term vision (including undecided issues and possible approaches), and to define the scope of work that could be done in the Swift 4 timeframe.

General Principles

Ergonomics

It's worth noting that ergonomics and correctness are mutually-reinforcing. An API that is easy to use--but incorrectly--cannot be considered an ergonomic success. Conversely, an API that's simply hard to use is also hard to use correctly. Achieving optimal performance without compromising ergonomics or correctness is a greater challenge.

Consistency with the Swift language and idioms is also important for ergonomics. There are several places both in the standard library and in the Foundation additions to `String` where patterns and practices found elsewhere could be applied to improve usability and familiarity.

API Surface Area

Primary data types such as `String` should have APIs that are easily understood given a signature and a one-line summary. Today, `String` fails that test. As you can see, the Standard Library and Foundation both contribute significantly to its overall complexity.

Method Arity	Standard Library	Foundation
0: `f()`	5	7
1: `f(:)`	19	48
2: `f(::)`	13	19
3: `f(:::)`	5	11
4: `f(::::)`	1	7
5: `f(:::::)`	-	2
6: `f(::::::)`	-	1

API Kind	Standard Library	Foundation
`init`	41	18
`func`	42	55
`subscript`	9	0
`var`	26	14

Total: 205 APIs

By contrast, `Int` has 80 APIs, none with more than two parameters.[0] String processing is complex enough; users shouldn't have to press through physical API sprawl just to get started.

Many of the choices detailed below contribute to solving this problem, including:

- Restoring `Collection` conformance and dropping the `.characters` view.
- Providing a more general, composable slicing syntax.
- Altering `Comparable` so that parameterized (e.g. case-insensitive) comparison fits smoothly into the basic syntax.
- Clearly separating language-dependent operations on text produced by and for humans from language-independent operations on text produced by and for machine processing.
- Relocating APIs that fall outside the domain of basic string processing and discouraging the proliferation of ad-hoc extensions.

Batteries Included

While `String` is available to all programs out-of-the-box, crucial APIs for basic string processing tasks are still inaccessible until `Foundation` is imported. While it makes sense that `Foundation` is needed for domain-specific jobs such as linguistic tagging, one should not need to import anything to, for example, do case-insensitive comparison.

Unicode Compliance and Platform Support

The Unicode standard provides a crucial objective reference point for what constitutes correct behavior in an extremely complex domain, so Unicode-correctness is, and will remain, a fundamental design principle behind Swift's `String`. That said, the Unicode standard is an evolving document, so this objective reference-point is not fixed.[1] While many of the most important operations--e.g. string hashing, equality, and non-localized comparison--will be stable, the semantics of others, such as grapheme breaking and localized comparison and case conversion, are expected to change as platforms are updated, so programs should be written so their correctness does not depend on precise stability of these semantics across OS versions or platforms. Although it may be possible to imagine static and/or dynamic analysis tools that will help users find such errors, the only sure way to deal with this fact of life is to educate users.

Design Points

Internationalization

There is strong evidence that developers cannot determine how to use internationalization APIs correctly. Although documentation could and should be improved, the sheer size, complexity, and diversity of these APIs is a major contributor to the problem, causing novices to tune out, and more experienced programmers to make avoidable mistakes.

The first step in improving this situation is to regularize all localized operations as invocations of normal string operations with extra parameters. Among other things, this means:

1. Doing away with `localizedXXX` methods.
2. Providing a terse way to name the current locale as a parameter.
3. Automatically adjusting defaults for options such as case sensitivity based on whether the operation is localized.
4. Removing correctness traps like `localizedCaseInsensitiveCompare` (see guidance in the Internationalization and Localization Guide).

Along with appropriate documentation updates, these changes will make localized operations more teachable, comprehensible, and approachable, thereby lowering a barrier that currently leads some developers to ignore localization issues altogether.

The Default Behavior of `String`

Although this isn't well-known, the most accessible form of many operations on Swift `String` (and `NSString`) are really only appropriate for text that is intended to be processed for, and consumed by, machines. The semantics of the operations with the simplest spellings are always non-localized and language-agnostic.

Two major factors play into this design choice:

1. Machine processing of text is important, so we should have first-class, accessible functions appropriate to that use case.

2. The most general localized operations require a locale parameter not required by their un-localized counterparts. This naturally skews complexity towards localized operations.

Reaffirming that `String`'s simplest APIs have language-independent/machine-processed semantics has the benefit of clarifying the proper default behavior of operations such as comparison, and allows us to make significant optimizations that were previously thought to conflict with Unicode.

Future Directions

One of the most common internationalization errors is the unintentional presentation to users of text that has not been localized, but regularizing APIs and improving documentation can go only so far in preventing this error. Combined with the fact that `String` operations are non-localized by default, the environment for processing human-readable text may still be somewhat error-prone in Swift 4.

For an audience of mostly non-experts, it is especially important that naïve code is very likely to be correct if it compiles, and that more sophisticated issues can be revealed progressively. For this reason, we intend to specifically and separately target localization and internationalization problems in the Swift 5 timeframe.

Operations With Options

There are three categories of common string operation that commonly need to be tuned in various dimensions:

Operation	Applicable Options
sort ordering	locale, case/diacritic/width-insensitivity
case conversion	locale
pattern matching	locale, case/diacritic/width-insensitivity

The defaults for case-, diacritic-, and width-insensitivity are sometimes different for localized operations than for non-localized operations, so for example a localized search should be case-insensitive by default, and a non-localized search should be case-sensitive by default. We propose a standard "language" of defaulted parameters to be used for these purposes, with usage roughly like this:

```
x.compared(to: y, case: .sensitive, in: swissGerman)

x.lowercased(in: .currentLocale)

x.allMatches(somePattern, case: .insensitive, diacritic: .insensitive)
```

This usage might be supported by code like this:

```
enum StringSensitivity {
  case sensitive
  case insensitive
}

extension Locale {
  static var currentLocale: Locale { ... }
}

extension Unicode {
  // An example of the option language in declaration context,
  // with nil defaults indicating unspecified, so defaults can be
  // driven by the presence/absence of a specific Locale
  func frobnicated(
    case caseSensitivity: StringSensitivity? = nil,
    diacritic diacriticSensitivity: StringSensitivity? = nil,
    width widthSensitivity: StringSensitivity? = nil,
    in locale: Locale? = nil
  ) -> Self { ... }
}
```

Comparing and Hashing Strings

Collation Semantics

What Unicode says about collation--which is used in `<`, `==`, and hashing-- turns out to be quite interesting, once you pick it apart. The full Unicode Collation Algorithm (UCA) works like this:

1. Fully normalize both strings.
2. Convert each string to a sequence of numeric triples to form a collation key.
3. "Flatten" the key by concatenating the sequence of first elements to the sequence of second elements to the sequence of third elements.
4. Lexicographically compare the flattened keys.

While step 1 can usually be done quickly and incrementally, step 2 uses a collation table that maps matching *sequences* of Unicode scalars in the normalized string to *sequences* of triples, which get accumulated into a collation key. Predictably, this is where the real costs lie.

However, there are some bright spots to this story. First, as it turns out, string sorting (localized or not) should be done down to what's called the "identical" level, which adds a step 3a: append the string's normalized form to the flattened collation key. At first blush this just adds work, but consider what it does for equality: two strings that normalize the same, naturally, will collate the same. But also, *strings that normalize differently will always collate differently*. In other words, for equality, it is sufficient to compare the strings' normalized forms and see if they are the same. We can therefore entirely skip the expensive part of collation for equality comparison.

Next, naturally, anything that applies to equality also applies to hashing: it is sufficient to hash the string's normalized form, bypassing collation keys. This should provide significant speedups over the current implementation. Perhaps more importantly, since comparison down to the "identical" level applies even to localized strings, it means that hashing and equality can be implemented exactly the same way for localized and non-localized text, and hash tables with localized keys will remain valid across current-locale changes.

Finally, once it is agreed that the *default* role for `String` is to handle machine-generated and machine-readable text, the default ordering of `String`s need no longer use the UCA at all. It is sufficient to order them in any way that's consistent with equality, so `String` ordering can simply be a lexicographical comparison of normalized forms,[4] (which is equivalent to lexicographically comparing the sequences of grapheme clusters), again bypassing step 2 and offering another speedup.

This leaves us executing the full UCA *only* for localized sorting, and ICU's implementation has apparently been very well optimized.

Following this scheme everywhere would also allow us to make sorting behavior consistent across platforms. Currently, we sort `String` according to the UCA, except that--*only on Apple platforms*--pairs of ASCII characters are ordered by unicode scalar value.

Syntax

Because the current `Comparable` protocol expresses all comparisons with binary operators, string comparisons--which may require additional options--do not fit smoothly into the existing syntax. At the same time, we'd like to solve other problems with comparison, as outlined in this proposal (implemented by changes at the head of this branch). We should adopt a modification of that proposal that uses a method rather than an operator `<=>`:

```
enum SortOrder { case before, same, after }

protocol Comparable : Equatable {
  func compared(to: Self) -> SortOrder
  ...
}
```

This change will give us a syntactic platform on which to implement methods with additional, defaulted arguments, thereby unifying and regularizing comparison across the library.

```
extension String {
  func compared(to: Self) -> SortOrder {
    ...
  }
}
```

Note: `SortOrder` should bridge to `NSComparisonResult`. It's also possible that the standard library simply adopts Foundation's `ComparisonResult` as is, but we believe the community should at least consider alternate naming before that happens. There will be an opportunity to discuss the choices in detail when the modified Comparison Proposal comes up for review.

`String` should be a `Collection` of `Character` s Again

In Swift 2.0, `String` 's `Collection` conformance was dropped, because we convinced ourselves that its semantics differed from those of `Collection` too significantly.

It was always well understood that if strings were treated as sequences of `UnicodeScalar` s, algorithms such as `lexicographicalCompare` , `elementsEqual` , and `reversed` would produce nonsense results. Thus, in Swift 1.0, `String` was a collection of `Character` (extended grapheme clusters). During 2.0 development, though, we realized that correct string concatenation could occasionally merge distinct grapheme clusters at the start and end of combined strings.

This quirk aside, every aspect of strings-as-collections-of-graphemes appears to comport perfectly with Unicode. We think the concatenation problem is tolerable, because the cases where it occurs all represent partially-formed constructs. The largest class--isolated combining characters such as ◌́ (U+0301 COMBINING ACUTE ACCENT)--are explicitly called out in the Unicode standard as "degenerate" or "defective". The other cases--such as a string ending in a zero-width joiner or half of a regional indicator--appear to be equally transient and unlikely outside of a text editor.

Admitting these cases encourages exploration of grapheme composition and is consistent with what appears to be an overall Unicode philosophy that "no special provisions are made to get marginally better behavior for... cases that never occur in practice." [2] Furthermore, it seems unlikely to disturb the semantics of any plausible algorithms. We can handle these cases by documenting them, explicitly stating that the elements of a `String` are an emergent property based on Unicode rules.

The benefits of restoring `Collection` conformance are substantial:

- Collection-like operations encourage experimentation with strings to investigate and understand their behavior. This is useful for teaching new programmers, but also good for experienced programmers who want to understand more about strings/unicode.

- Extended grapheme clusters form a natural element boundary for Unicode strings. For example, searching and matching operations will always produce results that line up on grapheme cluster boundaries.

- Character-by-character processing is a legitimate thing to do in many real use-cases, including parsing, pattern matching, and language-specific transformations such as transliteration.

- `Collection` conformance makes a wide variety of powerful operations available that are appropriate to `String` 's default role as the vehicle for machine processed text.

  The methods `String` would inherit from `Collection` , where similar to higher-level string algorithms, have the right semantics. For example, grapheme-wise `lexicographicalCompare` , `elementsEqual` , and application of `flatMap` with case-conversion, produce the same results one would expect from whole-string ordering comparison, equality comparison, and case-conversion, respectively. `reverse` operates correctly on graphemes, keeping diacritics moored to their base characters and leaving emoji intact. Other methods such as `index(of:)` and `contains` make obvious sense. A few `Collection` methods, like `min` and `max` , may not be particularly useful on `String` , but we don't consider that to be a problem worth solving, in the same way that we wouldn't try to suppress `min` and `max` on a `Set([UInt8])` that was used to store IP addresses.

- Many of the higher-level operations that we want to provide for `String`s, such as parsing and pattern matching, should apply to any `Collection`, and many of the benefits we want for `Collections`, such as unified slicing, should accrue equally to `String`. Making `String` part of the same protocol hierarchy allows us to write these operations once and not worry about keeping the benefits in sync.
- Slicing strings into substrings is a crucial part of the vocabulary of string processing, and all other sliceable things are `Collection`s. Because of its collection-like behavior, users naturally think of `String` in collection terms, but run into frustrating limitations where it fails to conform and are left to wonder where all the differences lie. Many simply "correct" this limitation by declaring a trivial conformance:

  ```
  extension String : BidirectionalCollection {}
  ```

  Even if we removed indexing-by-element from `String`, users could still do this:

  ```
  extension String : BidirectionalCollection {
    subscript(i: Index) -> Character { return characters[i] }
  }
  ```

  It would be much better to legitimize the conformance to `Collection` and simply document the oddity of any concatenation corner-cases, than to deny users the benefits on the grounds that a few cases are confusing.

Note that the fact that `String` is a collection of graphemes does *not* mean that string operations will necessarily have to do grapheme boundary recognition. See this section for details.

`Character` and `CharacterSet`

`Character`, which represents a Unicode extended grapheme cluster, is a bit of a black box, requiring conversion to `String` in order to do any introspection, including interoperation with ASCII. To fix this, we should:

- Add a `unicodeScalars` view much like `String`'s, so that the sub-structure of grapheme clusters is discoverable.
- Add a failable `init` from sequences of scalars (returning nil for sequences that contain 0 or 2+ graphemes).
- (Lower priority) expose some operations, such as `func uppercase() -> String`, `var isASCII: Bool`, and, to the extent they can be sensibly generalized, queries of Unicode properties that should also be exposed on `UnicodeScalar` such as `isAlphabetic` and `isGraphemeBase`.

Despite its name, `CharacterSet` currently operates on the Swift `UnicodeScalar` type. This means it is usable on `String`, but only by going through the unicode scalar view. To deal with this clash in the short term, `CharacterSet` should be renamed to `UnicodeScalarSet`. In the longer term, it may be appropriate to introduce a `CharacterSet` that provides similar functionality for extended grapheme clusters. [5]

Unification of Slicing Operations

Creating substrings is a basic part of string processing, but the slicing operations that we have in Swift are inconsistent in both their spelling and their naming:

- Slices with two explicit endpoints are done with subscript, and support in-place mutation:

```
s[i..<j].mutate()
```

- Slicing from an index to the end, or from the start to an index, is done with a method and does not support in-place mutation:

```
s.prefix(upTo: i).readOnly()
```

Prefix and suffix operations should be migrated to be subscripting operations with one-sided ranges i.e. `s.prefix(upTo: i)` should become `s[..<i]`, as in this proposal. With generic subscripting in the language, that will allow us to collapse a wide variety of methods and subscript overloads into a single implementation, and give users an easy-to-use and composable way to describe subranges.

Further extending this EDSL to integrate use-cases like `s.prefix(maxLength: 5)` is an ongoing research project that can be considered part of the potential long-term vision of text (and collection) processing.

Substrings

When implementing substring slicing, languages are faced with three options:

1. Make the substrings the same type as string, and share storage.
2. Make the substrings the same type as string, and copy storage when making the substring.
3. Make substrings a different type, with a storage copy on conversion to string.

We think number 3 is the best choice. A walk-through of the tradeoffs follows.

Same type, shared storage

In Swift 3.0, slicing a `String` produces a new `String` that is a view into a subrange of the original `String`'s storage. This is why `String` is 3 words in size (the start, length and buffer owner), unlike the similar `Array` type which is only one.

This is a simple model with big efficiency gains when chopping up strings into multiple smaller strings. But it does mean that a stored substring keeps the entire original string buffer alive even after it would normally have been released.

This arrangement has proven to be problematic in other programming languages, because applications sometimes extract small strings from large ones and keep those small strings long-term. That is considered a memory leak and was enough of a problem in Java that they changed from substrings sharing storage to making a copy in 1.7.

Same type, copied storage

Copying of substrings is also the choice made in C#, and in the default `NSString` implementation. This approach avoids the memory leak issue, but has obvious performance overhead in performing the copies.

This in turn encourages trafficking in string/range pairs instead of in substrings, for performance reasons, leading to API challenges. For example:

```
foo.compare(bar, range: start..<end)
```

Here, it is not clear whether `range` applies to `foo` or `bar`. This relationship is better expressed in Swift as a slicing operation:

```
foo[start..<end].compare(bar)
```

Not only does this clarify to which string the range applies, it also brings this sub-range capability to any API that operates on `String` "for free". So these other combinations also work equally well:

```
// apply range on argument rather than target
foo.compare(bar[start..<end])
// apply range on both
foo[start..<end].compare(bar[start1..<end1])
// compare two strings ignoring first character
foo.dropFirst().compare(bar.dropFirst())
```

In all three cases, an explicit range argument need not appear on the `compare` method itself. The implementation of `compare` does not need to know anything about ranges. Methods need only take range arguments when that was an integral part of their purpose (for example, setting the start and end of a user's current selection in a text box).

Different type, shared storage

The desire to share underlying storage while preventing accidental memory leaks occurs with slices of `Array`. For this reason we have an `ArraySlice` type. The inconvenience of a separate type is mitigated by most operations used on `Array` from the standard library being generic over `Sequence` or `Collection`.

We should apply the same approach for `String` by introducing a distinct `SubSequence` type, `Substring`. Similar advice given for `ArraySlice` would apply to `Substring`:

> *Important: Long-term storage of `Substring` instances is discouraged. A substring holds a reference to the entire storage of a larger string, not just to the portion it presents, even after the original string's lifetime ends. Long-term storage of a `Substring` may therefore prolong the lifetime of large strings that are no longer otherwise accessible, which can appear to be memory leakage.*

When assigning a `Substring` to a longer-lived variable (usually a stored property) explicitly of type `String`, a type conversion will be performed, and at this point the substring buffer is copied and the original string's storage can be released.

A `String` that was not its own `Substring` could be one word--a single tagged pointer--without requiring additional allocations. `Substring`s would be a view onto a `String`, so are 3 words - pointer to owner, pointer to start, and a length. The small string optimization for `Substring` would take advantage of the larger size, probably with a less compressed encoding for speed.

The downside of having two types is the inconvenience of sometimes having a `Substring` when you need a `String`, and vice-versa. It is likely this would be a significantly bigger problem than with `Array` and `ArraySlice`, as slicing of `String` is such a common operation. It is especially relevant to existing code that assumes `String` is the currency type -- that is, the default string type used for everyday exchange between APIs. To ease the pain of type mismatches, `Substring` should be a subtype of `String` in the same way that `Int` is a subtype of `Optional<Int>`. This would give users an implicit conversion from `Substring` to `String`, as well as the usual implicit conversions such as `[Substring]` to `[String]` that other subtype relationships receive.

In most cases, type inference combined with the subtype relationship should make the type difference a non-issue and users will not care which type they are using. For flexibility and optimizability, most operations from the standard library will traffic in generic models of `Unicode` .

Guidance for API Designers

In this model, **if a user is unsure about which type to use, `String` is always a reasonable default**. A `Substring` passed where `String` is expected will be implicitly copied. When compared to the "same type, copied storage" model, we have effectively deferred the cost of copying from the point where a substring is created until it must be converted to `String` for use with an API.

A user who needs to optimize away copies altogether should use this guideline: if for performance reasons you are tempted to add a `Range` argument to your method as well as a `String` to avoid unnecessary copies, you should instead use `Substring` .

The "Empty Subscript"

To make it easy to call such an optimized API when you only have a `String` (or to call any API that takes a `Collection` 's `SubSequence` when all you have is the `Collection`), we propose the following "empty subscript" operation,

```
extension Collection {
  subscript() -> SubSequence {
    return self[startIndex..<endIndex]
  }
}
```

which allows the following usage:

```
funcThatIsJustLooking(at: person.name[]) // pass person.name as Substring
```

The `[]` syntax can be offered as a fixit when needed, similar to `&` for an `inout` argument. While it doesn't help a user to convert `[String]` to `[Substring]` , the need for such conversions is extremely rare, can be done with a simple `map` (which could also be offered by a fixit):

```
takesAnArrayOfSubstring(arrayOfString.map { $0[] })
```

Other Options Considered

As we have seen, all three options above have downsides, but it's possible these downsides could be eliminated/mitigated by the compiler. We are proposing one such mitigation--implicit conversion--as part of the "different type, shared storage" option, to help avoid the cognitive load on developers of having to deal with a separate `Substring` type.

To avoid the memory leak issues of a "same type, shared storage" substring option, we considered whether the compiler could perform an implicit copy of the underlying storage when it detects the string is being "stored" for long term usage, say when it is assigned to a stored property. The trouble with this approach is it is very difficult for the compiler to distinguish between long-term storage versus short-term in the case of abstractions that rely on stored properties. For example, should the storing of a substring inside an `Optional` be considered long-term? Or the storing of multiple substrings inside an array? The latter would not work well in the case of a `components(separatedBy:)` implementation that intended to return an array of substrings. It would also be difficult to distinguish intentional medium-term storage of substrings, say by a lexer. There does not appear to be an effective consistent rule that could be applied in the general case for detecting when a substring is truly being stored long-term.

To avoid the cost of copying substrings under "same type, copied storage", the optimizer could be enhanced to reduce the impact of some of those copies. For example, this code could be optimized to pull the invariant substring out of the loop:

```
for _ in 0..<lots {
  someFunc(takingString: bigString[bigRange])
}
```

It's worth noting that a similar optimization is needed to avoid an equivalent problem with implicit conversion in the "different type, shared storage" case:

```
let substring = bigString[bigRange]
for _ in 0..<lots { someFunc(takingString: substring) }
```

However, in the case of "same type, copied storage" there are many use cases that cannot be optimized as easily. Consider the following simple definition of a recursive `contains` algorithm, which when substring slicing is linear makes the overall algorithm quadratic:

```
extension String {
  func containsChar(_ x: Character) -> Bool {
    return !isEmpty && (first == x || dropFirst().containsChar(x))
  }
}
```

For the optimizer to eliminate this problem is unrealistic, forcing the user to remember to optimize the code to not use string slicing if they want it to be efficient (assuming they remember):

```
extension String {
  // add optional argument tracking progress through the string
  func containsCharacter(_ x: Character, atOrAfter idx: Index? = nil) -> Bool {
    let idx = idx ?? startIndex
    return idx != endIndex
      && (self[idx] == x || containsCharacter(x, atOrAfter: index(after: idx)))
  }
}
```

Substrings, Ranges and Objective-C Interop

The pattern of passing a string/range pair is common in several Objective-C APIs, and is made especially awkward in Swift by the non-interchangeability of `Range<String.Index>` and `NSRange`.

```
s2.find(s2, sourceRange: NSRange(j..<s2.endIndex, in: s2))
```

In general, however, the Swift idiom for operating on a sub-range of a `Collection` is to *slice* the collection and operate on that:

```
s2.find(s2[j..<s2.endIndex])
```

Therefore, APIs that operate on an `NSString` / `NSRange` pair should be imported without the `NSRange` argument. The Objective-C importer should be changed to give these APIs special treatment so that when a `Substring` is passed, instead of being converted to a `String`, the full `NSString` and range are passed to the Objective-C method, thereby avoiding a copy.

As a result, you would never need to pass an `NSRange` to these APIs, which solves the impedance problem by eliminating the argument, resulting in more idiomatic Swift code while retaining the performance benefit. To help users manually handle any cases that remain, Foundation should be augmented to allow the following syntax for converting to and from `NSRange`:

```
let nsr = NSRange(i..<j, in: s) // An NSRange corresponding to s[i..<j]
let iToJ = Range(nsr, in: s)    // Equivalent to i..<j
```

The `Unicode` protocol

With `Substring` and `String` being distinct types and sharing almost all interface and semantics, and with the highest-performance string processing requiring knowledge of encoding and layout that the currency types can't provide, it becomes important to capture the common "string API" in a protocol. Since Unicode conformance is a key feature of string processing in Swift, we call that protocol `Unicode`:

Note: The following assumes several features that are planned but not yet implemented in Swift, and should be considered a sketch rather than a final design.

```
protocol Unicode
  : Comparable, BidirectionalCollection where Element == Character {

  associatedtype Encoding : UnicodeEncoding
  var encoding: Encoding { get }

  associatedtype CodeUnits
    : RandomAccessCollection where Element == Encoding.CodeUnit
  var codeUnits: CodeUnits { get }

  associatedtype UnicodeScalars
    : BidirectionalCollection where Element == UnicodeScalar
  var unicodeScalars: UnicodeScalars { get }

  associatedtype ExtendedASCII
    : BidirectionalCollection where Element == UInt32
  var extendedASCII: ExtendedASCII { get }

  var unicodeScalars: UnicodeScalars { get }
}

extension Unicode {
  // ... define high-level non-mutating string operations, e.g. search ...

  func compared<Other: Unicode>(
    to rhs: Other,
    case caseSensitivity: StringSensitivity? = nil,
    diacritic diacriticSensitivity: StringSensitivity? = nil,
    width widthSensitivity: StringSensitivity? = nil,
    in locale: Locale? = nil
  ) -> SortOrder { ... }
}

extension Unicode : RangeReplaceableCollection where CodeUnits :
  RangeReplaceableCollection {
  // Satisfy protocol requirement
  mutating func replaceSubrange<C : Collection>(_: Range<Index>, with: C)
    where C.Element == Element

  // ... define high-level mutating string operations, e.g. replace ...
}
```

The goal is that `Unicode` exposes the underlying encoding and code units in such a way that for types with a known representation (e.g. a high-performance `UTF8String`) that information can be known at compile-time and can be used to generate a single path, while still allowing types like `String` that admit multiple representations to use runtime queries and branches to fast path specializations.

Note: `Unicode` would make a fantastic namespace for much of what's in this proposal if we could get the ability to nest types and protocols in protocols.

Scanning, Matching, and Tokenization

Low-Level Textual Analysis

We should provide convenient APIs for processing strings by character. For example, it should be easy to cleanly express, "if this string starts with `"f"`, process the rest of the string as follows..." Swift is well-suited to expressing this common pattern beautifully, but we need to add the APIs. Here are two examples of the sort of code that might be possible given such APIs:

```
if let firstLetter = input.dropPrefix(alphabeticCharacter) {
  somethingWith(input) // process the rest of input
}

if let (number, restOfInput) = input.parsingPrefix(Int.self) {
  ...
}
```

The specific spelling and functionality of APIs like this are TBD. The larger point is to make sure matching-and-consuming jobs are well-supported.

Unified Pattern Matcher Protocol

Many of the current methods that do matching are overloaded to do the same logical operations in different ways, with the following axes:

- Logical Operation: `find`, `split`, `replace`, match at start.
- Kind of pattern: `CharacterSet`, `String`, a regex, a closure.

- Options, e.g. case/diacritic sensitivity, locale. Sometimes a part of the method name, and sometimes an argument.
- Whole string or subrange.

We should represent these aspects as orthogonal, composable components, abstracting pattern matchers into a protocol like this one, that can allow us to define logical operations once, without introducing overloads, and massively reducing API surface area.

For example, using the strawman prefix `%` syntax to turn string literals into patterns, the following pairs would all invoke the same generic methods:

```
if let found = s.firstMatch(%"searchString") { ... }
if let found = s.firstMatch(someRegex) { ... }

for m in s.allMatches((%"searchString"), case: .insensitive) { ... }
for m in s.allMatches(someRegex) { ... }

let items = s.split(separatedBy: ", ")
let tokens = s.split(separatedBy: CharacterSet.whitespace)
```

Note that, because Swift requires the indices of a slice to match the indices of the range from which it was sliced, operations like `firstMatch` can return a `Substring?` in lieu of a `Range<String.Index>?` : the indices of the match in the string being searched, if needed, can easily be recovered as the `startIndex` and `endIndex` of the `Substring` .

Note also that matching operations are useful for collections in general, and would fall out of this proposal:

```
// replace subsequences of contiguous NaNs with zero
forces.replace(oneOrMore([Float.nan]), [0.0])
```

Regular Expressions

Addressing regular expressions is out of scope for this proposal. That said, it is important to note that the pattern matching protocol mentioned above provides a suitable foundation for regular expressions, and types such as `NSRegularExpression` can easily be retrofitted to conform to it. In the future, support for regular expression literals in the compiler could allow for compile-time syntax checking and optimization.

String Indices

`String` currently has four views-- `characters` , `unicodeScalars` , `utf8` , and `utf16` --each with its own opaque index type. The APIs used to translate indices between views add needless complexity, and the opacity of indices makes them difficult to serialize.

The index translation problem has two aspects:

1. `String` views cannot consume one another's indices without a cumbersome conversion step. An index into a `String` 's `characters` must be translated before it can be used as a position in its `unicodeScalars` . Although these translations are rarely needed, they add conceptual and API complexity.
2. Many APIs in the core libraries and other frameworks still expose `String` positions as `Int` s and regions as `NSRange` s, which can only reference a `utf16` view and interoperate poorly with `String` itself.

Index Interchange Among Views

String's need for flexible backing storage and reasonably-efficient indexing (i.e. without dynamically allocating and reference-counting the indices themselves) means indices need an efficient underlying storage type. Although we do not wish to expose `String`'s indices *as* integers, `Int` offsets into underlying code unit storage makes a good underlying storage type, provided `String`'s underlying storage supports random-access. We think random-access *code-unit storage* is a reasonable requirement to impose on all `String` instances.

Making these `Int` code unit offsets conveniently accessible and constructible solves the serialization problem:

```
clipboard.write(s.endIndex.codeUnitOffset)
let offset = clipboard.read(Int.self)
let i = String.Index(codeUnitOffset: offset)
```

Index interchange between `String` and its `unicodeScalars`, `codeUnits`, and `extendedASCII` views can be made entirely seamless by having them share an index type (semantics of indexing a `String` between grapheme cluster boundaries are TBD--it can either trap or be forgiving). Having a common index allows easy traversal into the interior of graphemes, something that is often needed, without making it likely that someone will do it by accident.

- `String.index(after:)` should advance to the next grapheme, even when the index points partway through a grapheme.
- `String.index(before:)` should move to the start of the grapheme before the current position.

Seamless index interchange between `String` and its UTF-8 or UTF-16 views is not crucial, as the specifics of encoding should not be a concern for most use cases, and would impose needless costs on the indices of other views. That said, we can make translation much more straightforward by exposing simple bidirectional converting `init`s on both index types:

```
let u8Position = String.UTF8.Index(someStringIndex)
let originalPosition = String.Index(u8Position)
```

Index Interchange with Cocoa

We intend to address `NSRange`s that denote substrings in Cocoa APIs as described later in this document. That leaves the interchange of bare indices with Cocoa APIs trafficking in `Int`. Hopefully such APIs will be rare, but when needed, the following extension, which would be useful for all `Collections`, can help:

```
extension Collection {
  func index(offset: IndexDistance) -> Index {
    return index(startIndex, offsetBy: offset)
  }
  func offset(of i: Index) -> IndexDistance {
    return distance(from: startIndex, to: i)
  }
}
```

Then integers can easily be translated into offsets into a `String`'s `utf16` view for consumption by Cocoa:

```
let cocoaIndex = s.utf16.offset(of: String.UTF16Index(i))
let swiftIndex = s.utf16.index(offset: cocoaIndex)
```

Formatting

A full treatment of formatting is out of scope of this proposal, but we believe it's crucial for completing the text processing picture. This section details some of the existing issues and thinking that may guide future development.

Printf-Style Formatting

`String(format:)` is designed on the `printf` model: it takes a format string with textual placeholders for substitution, and an arbitrary list of other arguments. The syntax and meaning of these placeholders has a long history in C, but for anyone who doesn't use them regularly they are cryptic and complex, as the `printf (3)` man page attests.

Aside from complexity, this style of API has two major problems: First, the spelling of these placeholders must match up to the types of the arguments, in the right order, or the behavior is undefined. Some limited support for compile-time checking of this correspondence could be implemented, but only for the cases where the format string is a literal. Second, there's no reasonable way to extend the formatting vocabulary to cover the needs of new types: you are stuck with what's in the box.

Foundation Formatters

The formatters supplied by Foundation are highly capable and versatile, offering both formatting and parsing services. When used for formatting, though, the design pattern demands more from users than it should:

- Matching the type of data being formatted to a formatter type
- Creating an instance of that type
- Setting stateful options (`currency` , `dateStyle`) on the type. Note: the need for this step prevents the instance from being used and discarded in the same expression where it is created.
- Overall, introduction of needless verbosity into source

These may seem like small issues, but the experience of Apple localization experts is that the total drag of these factors on programmers is such that they tend to reach for `String(format:)` instead.

String Interpolation

Swift string interpolation provides a user-friendly alternative to printf's domain-specific language (just write ordinary swift code!) and its type safety problems (put the data right where it belongs!) but the following issues prevent it from being useful for localized formatting (among other jobs):

- SR-2303 We are unable to restrict types used in string interpolation.
- SR-1260 String interpolation can't distinguish (fragments of) the base string from the string substitutions.

In the long run, we should improve Swift string interpolation to the point where it can participate in most any formatting job. Mostly this centers around fixing the interpolation protocols per the previous item, and supporting localization.

To be able to use formatting effectively inside interpolations, it needs to be both lightweight (because it all happens in-situ) and discoverable. One approach would be to standardize on `format` methods, e.g.:

```
"Column 1: \(n.format(radix:16, width:8)) *** \(message)"

"Something with leading zeroes: \(x.format(fill: zero, width:8))"
```

C String Interop

Our support for interoperation with nul-terminated C strings is scattered and incoherent, with 6 ways to transform a C string into a `String` and four ways to do the inverse. These APIs should be replaced with the following

```swift
extension String {
  /// Constructs a `String` having the same contents as `nulTerminatedUTF8`.
  ///
  /// - Parameter nulTerminatedUTF8: a sequence of contiguous UTF-8 encoded
  ///   bytes ending just before the first zero byte (NUL character).
  init(cString nulTerminatedUTF8: UnsafePointer<CChar>)

  /// Constructs a `String` having the same contents as `nulTerminatedCodeUnits`.
  ///
  /// - Parameter nulTerminatedCodeUnits: a sequence of contiguous code units in
  ///   the given `encoding`, ending just before the first zero code unit.
  /// - Parameter encoding: describes the encoding in which the code units
  ///   should be interpreted.
  init<Encoding: UnicodeEncoding>(
    cString nulTerminatedCodeUnits: UnsafePointer<Encoding.CodeUnit>,
    encoding: Encoding)

  /// Invokes the given closure on the contents of the string, represented as a
  /// pointer to a null-terminated sequence of UTF-8 code units.
  func withCString<Result>(
    _ body: (UnsafePointer<CChar>) throws -> Result) rethrows -> Result
}
```

In both of the construction APIs, any invalid encoding sequence detected will have its longest valid prefix replaced by U+FFFD, the Unicode replacement character, per Unicode specification. This covers the common case. The replacement is done *physically* in the underlying storage and the validity of the result is recorded in the `String`'s `encoding` such that future accesses need not be slowed down by possible error repair separately.

Construction that is aborted when encoding errors are detected can be accomplished using APIs on the `encoding`. String types that retain their physical encoding even in the presence of errors and are repaired on-the-fly can be built as different instances of the `Unicode` protocol.

Unicode 9 Conformance

Unicode 9 (and MacOS 10.11) brought us support for family emoji, which changes the process of properly identifying `Character` boundaries. We need to update `String` to account for this change.

High-Performance String Processing

Many strings are short enough to store in 64 bits, many can be stored using only 8 bits per unicode scalar, others are best encoded in UTF-16, and some come to us already in some other encoding, such as UTF-8, that would be costly to translate. Supporting these formats while maintaining usability for general-purpose APIs demands that a single `String` type can be backed by many different representations.

That said, the highest performance code always requires static knowledge of the data structures on which it operates, and for this code, dynamic selection of representation comes at too high a cost. Heavy-duty text processing demands a way to opt out of dynamism and directly use known encodings. Having this ability can also make it easy to cleanly specialize code that handles dynamic cases for maximal efficiency on the most common representations.

To address this need, we can build models of the `Unicode` protocol that encode representation information into the type, such as `NFCNormalizedUTF16String`.

Parsing ASCII Structure

Although many machine-readable formats support the inclusion of arbitrary Unicode text, it is also common that their fundamental structure lies entirely within the ASCII subset (JSON, YAML, many XML formats). These formats are often processed most efficiently by recognizing ASCII structural elements as ASCII, and capturing the arbitrary sections between them in more-general strings. The current String API offers no way to efficiently recognize ASCII and skip past everything else without the overhead of full decoding into unicode scalars.

For these purposes, strings should supply an `extendedASCII` view that is a collection of `UInt32`, where values less than `0x80` represent the corresponding ASCII character, and other values represent data that is specific to the underlying encoding of the string.

Language Support

This proposal depends on two new features in the Swift language:

1. **Generic subscripts**, to enable unified slicing syntax.
2. **A subtype relationship** between `Substring` and `String`, enabling framework APIs to traffic solely in `String` while still making it possible to avoid copies by handling `Substring`s where necessary.

Additionally, **the ability to nest types and protocols inside protocols** could significantly shrink the footprint of this proposal on the top-level Swift namespace.

Open Questions

Must `String` be limited to storing UTF-16 subset encodings?

- The ability to handle `UTF-8`-encoded strings (models of `Unicode`) is not in question here; this is about what encodings must be storable, without transcoding, in the common currency type called "`String`".
- ASCII, Latin-1, UCS-2, and UTF-16 are UTF-16 subsets. UTF-8 is not.
- If we have a way to get at a `String`'s code units, we need a concrete type in which to express them in the API of `String`, which is a concrete type
- If String needs to be able to represent UTF-32, presumably the code units need to be `UInt32`.
- Not supporting UTF-32-encoded text seems like one reasonable design choice.
- Maybe we can allow UTF-8 storage in `String` and expose its code units as `UInt16`, just as we would for Latin-1.
- Supporting only UTF-16-subset encodings would imply that `String` indices can be serialized without recording the `String`'s underlying encoding.

Do we need a type-erasable base protocol for UnicodeEncoding?

UnicodeEncoding has an associated type, but it may be important to be able to traffic in completely dynamic encoding values, e.g. for "tell me the most efficient encoding for this string."

Should there be a string "facade?"

One possible design alternative makes `Unicode` a vehicle for expressing the storage and encoding of code units, but does not attempt to give it an API appropriate for `String`. Instead, string APIs would be provided by a generic wrapper around an instance of `Unicode`:

```swift
struct StringFacade<U: Unicode> : BidirectionalCollection {

  // ...APIs for high-level string processing here...

  var unicode: U // access to lower-level unicode details
}

typealias String = StringFacade<StringStorage>
typealias Substring = StringFacade<StringStorage.SubSequence>
```

This design would allow us to de-emphasize lower-level `String` APIs such as access to the specific encoding, by putting them behind a `.unicode` property. A similar effect in a facade-less design would require a new top-level `StringProtocol` playing the role of the facade with an `associatedtype Storage : Unicode`.

An interesting variation on this design is possible if defaulted generic parameters are introduced to the language:

```swift
struct String<U: Unicode = StringStorage>
  : BidirectionalCollection {

  // ...APIs for high-level string processing here...

  var unicode: U // access to lower-level unicode details
}

typealias Substring = String<StringStorage.SubSequence>
```

One advantage of such a design is that naïve users will always extend "the right type" (`String`) without thinking, and the new APIs will show up on `Substring`, `MyUTF8String`, etc. That said, it also has downsides that should not be overlooked, not least of which is the confusability of the meaning of the word "string." Is it referring to the generic or the concrete type?

`TextOutputStream` and `TextOutputStreamable`

`TextOutputStreamable` is intended to provide a vehicle for efficiently transporting formatted representations to an output stream without forcing the allocation of storage. Its use of `String`, a type with multiple representations, at the lowest-level unit of communication, conflicts with this goal. It might be sufficient to change `TextOutputStream` and `TextOutputStreamable` to traffic in an associated type conforming to `Unicode`, but that is not yet clear. This area will require some design work.

`description` and `debugDescription`

- Should these be creating localized or non-localized representations?
- Is returning a `String` efficient enough?
- Is `debugDescription` pulling the weight of the API surface area it adds?

`StaticString`

`StaticString` was added as a byproduct of standard library development and kept around because it seemed useful, but it was never truly *designed* for client programmers. We need to decide what happens with it. Presumably *something* should fill its role, and that should conform to `Unicode`.

Footnotes

0 The integers rewrite currently underway is expected to substantially reduce the scope of `Int`'s API by using more generics. ↩

1 In practice, these semantics will usually be tied to the version of the installed ICU library, which programmatically encodes the most complex rules of the Unicode Standard and its de-facto extension, CLDR.↩

2 See http://unicode.org/reports/tr29/#Notation. Note that inserting Unicode scalar values to prevent merging of grapheme clusters would also constitute a kind of misbehavior (one of the clusters at the boundary would not be found in the result), so would be relatively costly to implement, with little benefit. ↩

4 The use of non-UCA-compliant ordering is fully sanctioned by the Unicode standard for this purpose. In fact there's a whole chapter dedicated to it. In particular, §5.17 says:

> *When comparing text that is visible to end users, a correct linguistic sort should be used, as described in Section 5.16, Sorting and Searching. However, in many circumstances the only requirement is for a fast, well-defined ordering. In such cases, a binary ordering can be used.*

↩

5 The queries supported by `NSCharacterSet` map directly onto properties in a table that's indexed by unicode scalar value. This table is part of the Unicode standard. Some of these queries (e.g., "is this an uppercase character?") may have fairly obvious generalizations to grapheme clusters, but exactly how to do it is a research topic and *ideally* we'd either establish the existing practice that the Unicode committee would standardize, or the Unicode committee would do the research and we'd implement their result.↩