Literals

What happens when a literal expression is used?

The complicated case is for integer, floating-point, character, and string literals, so let's look at those.

High-Level View

```
window.setTitle("Welcome to Xcode")
```

In this case, we have a string literal and an enclosing context. If `window` is an NSWindow, there will only be one possible method named `setTitle`, which takes an NSString. Therefore, we want the string literal expression to end up being an NSString.

Fortunately, NSString implements ExpressibleByStringLiteral, so the type checker will indeed be able to choose NSString as the type of the string literal. All is well.

In the case of integers or floating-point literals, the value effectively has infinite precision. Once the type has been chosen, the value is checked to see if it is in range for that type.

The ExpressibleByStringLiteral Protocol

Here is the `ExpressibleByStringLiteral` protocol as defined in the standard library's CompilerProtocols.swift::

```
/// A type that can be initialized with a string literal.
///
/// The `String` and `StaticString` types conform to the
/// `ExpressibleByStringLiteral` protocol. You can initialize a variable or
/// constant of either of these types using a string literal of any length.
///
///     let picnicGuest = "Deserving porcupine"
///
/// Conforming to ExpressibleByStringLiteral
/// ========================================
///
/// To add `ExpressibleByStringLiteral` conformance to your custom type,
/// implement the required initializer.
public protocol ExpressibleByStringLiteral
  : ExpressibleByExtendedGraphemeClusterLiteral {

  /// A type that represents a string literal.
  ///
  /// Valid types for `StringLiteralType` are `String` and `StaticString`.
  associatedtype StringLiteralType: _ExpressibleByBuiltinStringLiteral

  /// Creates an instance initialized to the given string value.
  ///
  /// - Parameter value: The value of the new instance.
```

```
  init(stringLiteral value: StringLiteralType)
}
```

Curiously, the protocol is not defined in terms of primitive types, but in terms of any StringLiteralType that the implementer chooses. In most cases, this will be Swift's own native String type, which means users can implement their own ExpressibleByStringLiteral types while still dealing with a high-level interface.

(Why is this not hardcoded? A String *must* be a valid Unicode string, but if the string literal contains escape sequences, an invalid series of code points could be constructed...which may be what's desired in some cases.)

The _ExpressibleByBuiltinStringLiteral Protocol

`CompilerProtocols.swift` contains a second protocol::

```
// NOTE: the compiler has builtin knowledge of this protocol
public protocol _ExpressibleByBuiltinStringLiteral
  : _ExpressibleByBuiltinExtendedGraphemeClusterLiteral {

  init(
      _builtinStringLiteral start: Builtin.RawPointer,
      utf8CodeUnitCount: Builtin.Word,
      isASCII: Builtin.Int1)
}
```

The use of builtin types makes it clear that this is *only* for use in the standard library. This is the actual primitive function that is used to construct types from string literals: the compiler knows how to emit raw data from the literal, and the arguments describe that raw data.

So, the general runtime behavior is now clear:

1. The compiler generates raw string data.
2. Some type conforming to _ExpressibleByBuiltinStringLiteral is constructed from the raw string data. This will be a standard library type.
3. Some type conforming to ExpressibleByStringLiteral is constructed from the object constructed in step 2. This may be a user-defined type. This is the result.

The Type-Checker's Algorithm

In order to make this actually happen, the type-checker has to do some fancy footwork. Remember, at this point all we have is a string literal and an expected type; if the function were overloaded, we would have to try all the types.

This algorithm can go forwards or backwards, since it's actually defined in terms of constraints, but it's easiest to understand as a linear process.

1. Filter the types provided by the context to only include those that are ExpressibleByStringLiteral.
2. Using the associated StringLiteralType, find the appropriate `_convertFromBuiltinStringLiteral`.
3. Using the type from step 1, find the appropriate `convertFromStringLiteral`.
4. Build an expression tree with the appropriate calls.

How about cases where there is no context? ::

var str = "abc"

Here we have nothing to go on, so instead the type checker looks for a global type named `StringLiteralType` in the current module-scope context, and uses that type if it is actually a ExpressibleByStringLiteral type. This both allows different standard libraries to set different default literal types, and allows a user to *override* the default type in their own source file.

The real story is even more complicated because of implicit conversions: the type expected by `setTitle` might not actually be literal-convertible, but something else that *is* literal-convertible can then implicitly convert to the proper type. If this makes your head spin, don't worry about it.

Arrays, Dictionaries, and Interpolation

Array and dictionary literals don't have a Builtin*Convertible form. Instead, they just always use a variadic list of elements (`T...`) in the array case and (key, value) tuples in the dictionary case. A variadic list is always exposed using the standard library's Array type, so there is no separate step to jump through.

The default array literal type is always Array, and the default dictionary literal type is always Dictionary.

String interpolations are a bit different: they create an instance of `T.StringInterpolation` and append each segment to it, then initialize an instance of `T` with that instance. The default type for an interpolated literal without context is also `StringLiteralType` .