orphan:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\ABI\(swift-main)(docs)(ABI)Mangling.rst`, line 5)

Unknown directive type "highlight".

```
.. highlight:: none
```

Mangling

```
mangled-name ::= '$s' global  // Swift stable mangling
mangled-name ::= '_T0' global // Swift 4.0
mangled-name ::= '$S' global  // Swift 4.2
```

All Swift-mangled names begin with a common prefix. Since Swift 4.0, the compiler has used variations of the mangling described in this document, though pre-stable versions may not exactly conform to this description. By using distinct prefixes, tools can attempt to accommodate bugs and version variations in pre-stable versions of Swift.

The basic mangling scheme is a list of 'operators' where the operators are structured in a post-fix order. For example the mangling may start with an identifier but only later in the mangling a type-like operator defines how this identifier has to be interpreted:

```
4Test3FooC   // The trailing 'C' says that 'Foo' is a class in module 'Test'
```

Operators are either identifiers or a sequence of one or more characters, like `C` for class. All operators share the same name-space. Important operators are a single character, which means that no other operator may start with the same character.

Some less important operators are longer and may also contain one or more natural numbers. But it's always important that the demangler can identify the end (the last character) of an operator. For example, it's not possible to determine the last character if there are two operators `M` and `Ma`: `a` could belong to `M` or it could be the first character of the next operator.

The intention of the post-fix order is to optimize for common pre-fixes. Regardless, if it's the mangling for a metatype or a function in a module, the mangled name will start with the module name (after the `_S`).

In the following, productions which are only _part_ of an operator, are named with uppercase letters.

Symbolic references

The Swift compiler emits mangled names into binary images to encode references to types for runtime instantiation and reflection. In a binary, these mangled names may embed pointers to runtime data structures in order to more efficiently represent locally-defined types. We call these pointers **symbolic references**. These references will be introduced by a control character in the range *x01* ... *x1F*, which indicates the kind of symbolic reference, followed by some number of arbitrary bytes *which may include null bytes*. Code that processes mangled names out of Swift binaries needs to be aware of symbolic references in order to properly terminate strings; a null terminator may be part of a symbolic reference.

```
symbolic-reference ::= [\x01-\x17] .{4} // Relative symbolic reference
 #if sizeof(void*) == 8
   symbolic-reference ::= [\x18-\x1F] .{8} // Absolute symbolic reference
 #elif sizeof(void*) == 4
   symbolic-reference ::= [\x18-\x1F] .{4} // Absolute symbolic reference
 #endif
```

Symbolic references are only valid in compiler-emitted metadata structures and must only appear in read-only parts of a binary image. APIs and tools that interpret Swift mangled names from potentially uncontrolled inputs must refuse to interpret symbolic references.

The following symbolic reference kinds are currently implemented:

```
#if SWIFT_RUNTIME_VERSION < 5.1
  {any-generic-type, protocol} ::= '\x01' .{4} // Reference points directly to context descriptor
  {any-generic-type, protocol} ::= '\x02' .{4} // Reference points indirectly to context descriptor
#else
  {any-generic-type, protocol, opaque-type-decl-name} ::= '\x01' .{4} // Reference points directly to context descriptor
  {any-generic-type, protocol, opaque-type-decl-name} ::= '\x02' .{4} // Reference points indirectly to context descriptor
#endif
// The grammatical role of the symbolic reference is determined by the
// kind of context descriptor referenced

protocol-conformance-ref ::= '\x03' .{4}  // Reference points directly to protocol conformance descriptor (NOT IMPLEMENTED)
protocol-conformance-ref ::= '\x04' .{4}  // Reference points indirectly to protocol conformance descriptor (NOT IMPLEMENTED)

dependent-associated-conformance ::= '\x05' .{4}  // Reference points directly to associated conformance descriptor (NOT IMPLEMENTED)
dependent-associated-conformance ::= '\x06' .{4}  // Reference points indirectly to associated conformance descriptor (NOT IMPLEMENTED)

associated-conformance-access-function ::= '\x07' .{4}  // Reference points directly to associated conformance access function relative t
associated-conformance-access-function ::= '\x08' .{4}  // Reference points directly to associated conformance access function relative t

// keypaths only in Swift 5.0, generalized in Swift 5.1
#if SWIFT_RUNTIME_VERSION >= 5.1
  metadata-access-function ::= '\x09' .{4}  // Reference points directly to metadata access function that can be invoked to produce refer
#endif
```

A mangled name may also include `\xFF` bytes, which are only used for alignment padding. They do not affect what the mangled name references and can be skipped over and ignored.

Globals

```
global ::= type 'N'                    // type metadata (address point)
                                       // -- type starts with [BCOSTV]
global ::= type 'Mf'                   // 'full' type metadata (start of object)
global ::= type 'MP'                   // type metadata pattern
global ::= type 'Ma'                   // type metadata access function
global ::= type 'ML'                   // type metadata lazy cache variable
global ::= nominal-type 'Mr'           // generic type completion function
global ::= nominal-type 'Mi'           // generic type instantiation function
global ::= nominal-type 'MI'           // generic type instantiation cache
global ::= nominal-type 'Ml'           // in-place type initialization cache
global ::= nominal-type 'Mm'           // class metaclass
global ::= nominal-type 'Mn'           // nominal type descriptor
#if SWIFT_RUNTIME_VERSION >= 5.1
  global ::= opaque-type-decl-name 'MQ'  // opaque type descriptor -- added in Swift 5.1
#endif
global ::= nominal-type 'Mu'           // class method lookup function
global ::= nominal-type 'MU'           // ObjC metadata update callback function
global ::= nominal-type 'Ms'           // ObjC resilient class stub
global ::= nominal-type 'Mt'           // Full ObjC resilient class stub (private)
global ::= module 'MXM'                // module descriptor
global ::= context 'MXE'               // extension descriptor
global ::= context 'MXX'               // anonymous context descriptor
global ::= context identifier 'MXY'    // anonymous context descriptor
global ::= type assoc-type-list 'MXA'  // generic parameter ref (HISTORICAL)
global ::= protocol 'Mp'               // protocol descriptor

global ::= protocol 'Hr'               // protocol descriptor runtime record
global ::= nominal-type 'Hn'           // nominal type descriptor runtime record
#if SWIFT_RUNTIME_VERSION >= 5.1
  global ::= opaque-type 'Ho'          // opaque type descriptor runtime record
#endif
global ::= protocol-conformance 'Hc'   // protocol conformance runtime record
global ::= global 'HF'                 // accessible function runtime record

global ::= nominal-type 'Mo'           // class metadata immediate member base offset

global ::= type 'MF'                   // metadata for remote mirrors: field descriptor
global ::= type 'MB'                   // metadata for remote mirrors: builtin type descriptor
```

```
global ::= protocol-conformance 'MA'   // metadata for remote mirrors: associated type descriptor
global ::= nominal-type 'MC'           // metadata for remote mirrors: superclass descriptor

// TODO check this::
global ::= mangled-name 'TA'                     // partial application forwarder
global ::= mangled-name 'Ta'                     // ObjC partial application forwarder
global ::= mangled-name 'TQ' index               // Async await continuation partial function
global ::= mangled-name 'TY' index               // Async suspend continuation partial function

global ::= type 'w' VALUE-WITNESS-KIND // value witness

global ::= protocol 'MS'               // protocol self-conformance descriptor
global ::= protocol 'WS'               // protocol self-conformance witness table
global ::= protocol-conformance 'Mc'   // protocol conformance descriptor
global ::= protocol-conformance 'WP'   // protocol witness table
global ::= protocol-conformance 'Wa'   // protocol witness table accessor (HISTORICAL)

global ::= protocol-conformance 'WG'   // generic protocol witness table (HISTORICAL)
global ::= protocol-conformance 'Wp'   // protocol witness table pattern
global ::= protocol-conformance 'Wr'   // resilient witness table (HISTORICAL)
global ::= protocol-conformance 'WI'   // generic protocol witness table instantiation function
global ::= type protocol-conformance 'WL'   // lazy protocol witness table cache variable

global ::= protocol-conformance identifier 'Wt' // associated type metadata accessor (HISTORICAL)
global ::= protocol-conformance assoc-type-list protocol 'WT' // associated type witness table accessor
global ::= protocol-conformance protocol 'Wb' // base protocol witness table accessor
global ::= type protocol-conformance 'Wl' // lazy protocol witness table accessor

global ::= global generic-signature? 'WJ' DIFFERENTIABILITY-KIND INDEX-SUBSET 'p' INDEX-SUBSET 'r' // differentiability witness

global ::= type 'WV'                   // value witness table
global ::= entity 'Wvd'                // field offset
global ::= entity 'WC'                 // resilient enum tag index

global ::= global 'MK'                 // instantiation cache associated with global

global ::= global 'MJ'                 // noncanonical specialized generic type metadata instantiation cache associated with global
global ::= global 'MN'                 // noncanonical specialized generic type metadata for global
global ::= global 'Mz'                 // canonical specialized generic type metadata caching token

#if SWIFT_RUNTIME_VERSION >= 5.4
  global ::= context (decl-name '_')+ 'WZ' // global variable one-time initialization function
  global ::= context (decl-name '_')+ 'Wz' // global variable one-time initialization token
#endif
```

A direct symbol resolves directly to the address of an object. An indirect symbol resolves to the address of a pointer to the object. They are distinct manglings to make a certain class of bugs immediately obvious.

The terminology is slightly overloaded when discussing offsets. A direct offset resolves to a variable holding the true offset. An indirect offset resolves to a variable holding an offset to be applied to type metadata to get the address of the true offset. (Offset variables are required when the object being accessed lies within a resilient structure. When the layout of the object may depend on generic arguments, these offsets must be kept in metadata. Indirect field offsets are therefore required when accessing fields in generic types where the metadata itself has unknown layout.)

```
global ::= global 'Tj'                 // resilient method dispatch thunk
global ::= global 'Tq'                 // method descriptor

global ::= global 'TO'                 // ObjC-as-swift thunk
global ::= global 'To'                 // swift-as-ObjC thunk
global ::= global 'TD'                 // dynamic dispatch thunk
global ::= global 'Td'                 // direct method reference thunk
global ::= global 'TE'                 // distributed actor thunk
global ::= global 'TF'                 // distributed method accessor
global ::= global 'TI'                 // implementation of a dynamic_replaceable function
global ::= global 'Tu'                 // async function pointer of a function
global ::= global 'TX'                 // function pointer of a dynamic_replaceable function
global ::= global 'Twb'                // back deployment thunk
global ::= global 'TwB'                // back deployment fallback function
global ::= entity entity 'TV'          // vtable override thunk, derived followed by base
global ::= type label-list? 'D'        // type mangling for the debugger with label list for function types.
global ::= type 'TC'                   // continuation prototype (not actually used for real symbols)
global ::= protocol-conformance entity 'TW' // protocol witness thunk
global ::= entity 'TS'                 // protocol self-conformance witness thunk
global ::= context identifier identifier 'TB' // property behavior initializer thunk (not used currently)
global ::= context identifier identifier 'Tb' // property behavior setter thunk (not used currently)
global ::= global specialization       // function specialization
global ::= global 'Tm'                 // merged function
global ::= entity                      // some identifiable thing
global ::= from-type to-type generic-signature? 'TR'  // reabstraction thunk
global ::= impl-function-type type 'Tz' index? // objc-to-swift-async completion handler block implementation
global ::= impl-function-type type 'TZ' index? // objc-to-swift-async completion handler block implementation (predefined by runtime)
global ::= from-type to-type generic-signature? 'TR'  // reabstraction thunk
global ::= impl-function-type type generic-signature? 'Tz'     // objc-to-swift-async completion handler block implementation
global ::= impl-function-type type generic-signature? 'TZ'     // objc-to-swift-async completion handler block implementation (predefined
global ::= from-type to-type self-type generic-signature? 'Ty'  // reabstraction thunk with dynamic 'Self' capture
global ::= from-type to-type generic-signature? 'Tr'  // obsolete mangling for reabstraction thunk
global ::= entity generic-signature? type type* 'TK' // key path getter
global ::= entity generic-signature? type type* 'Tk' // key path setter
global ::= type generic-signature 'TH' // key path equality
global ::= type generic-signature 'Th' // key path hasher
global ::= global generic-signature? 'TJ' AUTODIFF-FUNCTION-KIND INDEX-SUBSET 'p' INDEX-SUBSET 'r' // autodiff function
global ::= global generic-signature? 'TJV' AUTODIFF-FUNCTION-KIND INDEX-SUBSET 'p' INDEX-SUBSET 'r' // autodiff derivative vtable thunk
global ::= from-type to-type 'TJO' AUTODIFF-FUNCTION-KIND // autodiff self-reordering reabstraction thunk
global ::= from-type 'TJS' AUTODIFF-FUNCTION-KIND INDEX-SUBSET 'p' INDEX-SUBSET 'r' INDEX-SUBSET 'P' // autodiff linear map subset parame
global ::= global to-type 'TJS' AUTODIFF-FUNCTION-KIND INDEX-SUBSET 'p' INDEX-SUBSET 'r' INDEX-SUBSET 'P' // autodiff derivative functior

global ::= protocol 'TL'               // protocol requirements base descriptor
global ::= assoc-type-name 'Tl'        // associated type descriptor
global ::= assoc-type-name 'TM'        // default associated type witness accessor (HISTORICAL)
global ::= type assoc-type-list protocol 'Tn' // associated conformance descriptor
global ::= type assoc-type-list protocol 'TN' // default associated conformance witness accessor
global ::= type protocol 'Tb'          // base conformance descriptor

REABSTRACT-THUNK-TYPE ::= 'R'          // reabstraction thunk
REABSTRACT-THUNK-TYPE ::= 'r'          // reabstraction thunk (obsolete)

global ::= reabstraction-thunk type 'TU' // reabstraction thunk with global actor constraint
```

The *from-type* and *to-type* in a reabstraction thunk helper function are always non-polymorphic `<impl-function-type>` types.

```
VALUE-WITNESS-KIND ::= 'al'           // allocateBuffer
VALUE-WITNESS-KIND ::= 'ca'           // assignWithCopy
VALUE-WITNESS-KIND ::= 'ta'           // assignWithTake
VALUE-WITNESS-KIND ::= 'de'           // deallocateBuffer
VALUE-WITNESS-KIND ::= 'xx'           // destroy
VALUE-WITNESS-KIND ::= 'XX'           // destroyBuffer
VALUE-WITNESS-KIND ::= 'Xx'           // destroyArray
VALUE-WITNESS-KIND ::= 'CP'           // initializeBufferWithCopyOfBuffer
VALUE-WITNESS-KIND ::= 'Cp'           // initializeBufferWithCopy
VALUE-WITNESS-KIND ::= 'cp'           // initializeWithCopy
VALUE-WITNESS-KIND ::= 'TK'           // initializeBufferWithTakeOfBuffer
VALUE-WITNESS-KIND ::= 'Tk'           // initializeBufferWithTake
VALUE-WITNESS-KIND ::= 'tk'           // initializeWithTake
VALUE-WITNESS-KIND ::= 'pr'           // projectBuffer
VALUE-WITNESS-KIND ::= 'xs'           // storeExtraInhabitant
VALUE-WITNESS-KIND ::= 'xg'           // getExtraInhabitantIndex
VALUE-WITNESS-KIND ::= 'Cc'           // initializeArrayWithCopy
VALUE-WITNESS-KIND ::= 'Tt'           // initializeArrayWithTakeFrontToBack
VALUE-WITNESS-KIND ::= 'tT'           // initializeArrayWithTakeBackToFront
VALUE-WITNESS-KIND ::= 'ug'           // getEnumTag
```

```
VALUE-WITNESS-KIND ::= 'up'           // destructiveProjectEnumData
VALUE-WITNESS-KIND ::= 'ui'           // destructiveInjectEnumTag
```

<VALUE-WITNESS-KIND> differentiates the kinds of value witness functions for a type.

```
AUTODIFF-FUNCTION-KIND ::= 'f'        // JVP (forward-mode derivative)
AUTODIFF-FUNCTION-KIND ::= 'r'        // VJP (reverse-mode derivative)
AUTODIFF-FUNCTION-KIND ::= 'd'        // differential
AUTODIFF-FUNCTION-KIND ::= 'p'        // pullback
```

<AUTODIFF-FUNCTION-KIND> differentiates the kinds of functions assocaited with a differentiable function used for differentiable programming.

```
global ::= generic-signature? type 'WOy' // Outlined copy
global ::= generic-signature? type 'WOe' // Outlined consume
global ::= generic-signature? type 'WOr' // Outlined retain
global ::= generic-signature? type 'WOs' // Outlined release
global ::= generic-signature? type 'WOb' // Outlined initializeWithTake
global ::= generic-signature? type 'WOc' // Outlined initializeWithCopy
global ::= generic-signature? type 'WOd' // Outlined assignWithTake
global ::= generic-signature? type 'WOf' // Outlined assignWithCopy
global ::= generic-signature? type 'WOh' // Outlined destroy
```

Entities

```
entity ::= nominal-type                    // named type declaration
entity ::= context entity-spec static? curry-thunk?

static ::= 'Z'
curry-thunk ::= 'Tc'

label-list ::= empty-list            // represents complete absence of parameter labels
label-list ::= ('_' | identifier)*   // '_' is inserted as placeholder for empty label,
                                     // since the number of labels should match the number of parameters

// The leading type is the function type
entity-spec ::= label-list type file-discriminator? 'fC'      // allocating constructor
entity-spec ::= label-list type file-discriminator? 'fc'      // non-allocating constructor
entity-spec ::= type 'fU' INDEX            // explicit anonymous closure expression
entity-spec ::= type 'fu' INDEX            // implicit anonymous closure
entity-spec ::= 'fA' INDEX                 // default argument N+1 generator
entity-spec ::= 'fi'                       // non-local variable initializer
entity-spec ::= 'fP'                       // property wrapper backing initializer
entity-spec ::= 'fW'                       // property wrapper init from projected value
entity-spec ::= 'fD'                       // deallocating destructor; untyped
entity-spec ::= 'fd'                       // non-deallocating destructor; untyped
entity-spec ::= 'fE'                       // ivar destroyer; untyped
entity-spec ::= 'fe'                       // ivar initializer; untyped
entity-spec ::= 'Tv' NATURAL               // outlined global variable (from context function)
entity-spec ::= 'Te' bridge-spec           // outlined objective c method call

entity-spec ::= decl-name label-list function-signature generic-signature? 'F'    // function
entity-spec ::= label-list type file-discriminator? 'i' ACCESSOR                  // subscript
entity-spec ::= decl-name label-list? type 'v' ACCESSOR                           // variable
entity-spec ::= decl-name type 'fp'                                               // generic type parameter
entity-spec ::= decl-name type 'fo'                                               // enum element (currently not used)
entity-spec ::= identifier 'Qa'                                                   // associated type declaration

ACCESSOR ::= 'm'                           // materializeForSet
ACCESSOR ::= 's'                           // setter
ACCESSOR ::= 'g'                           // getter
ACCESSOR ::= 'G'                           // global getter
ACCESSOR ::= 'w'                           // willSet
ACCESSOR ::= 'W'                           // didSet
ACCESSOR ::= 'r'                           // read
ACCESSOR ::= 'M'                           // modify (temporary)
ACCESSOR ::= 'a' ADDRESSOR-KIND            // mutable addressor
ACCESSOR ::= 'l' ADDRESSOR-KIND            // non-mutable addressor
ACCESSOR ::= 'p'                           // pseudo accessor referring to the storage itself

ADDRESSOR-KIND ::= 'u'                     // unsafe addressor (no owner)
ADDRESSOR-KIND ::= 'O'                     // owning addressor (non-native owner), not used anymore
ADDRESSOR-KIND ::= 'o'                     // owning addressor (native owner), not used anymore
ADDRESSOR-KIND ::= 'p'                     // pinning addressor (native owner), not used anymore

decl-name ::= identifier
decl-name ::= identifier 'L' INDEX                  // locally-discriminated declaration
decl-name ::= identifier identifier 'LL'            // file-discriminated declaration
decl-name ::= identifier 'L' RELATED-DISCRIMINATOR  // related declaration

RELATED-DISCRIMINATOR ::= [a-j]
RELATED-DISCRIMINATOR ::= [A-J]

file-discriminator ::= identifier 'Ll'     // anonymous file-discriminated declaration
```

The identifier in a <file-discriminator> and the second identifier in a file-discriminated <decl-name> is a string that represents the file the original declaration came from. It should be considered unique within the enclosing module. The first identifier is the name of the entity. Not all declarations marked `private` declarations will use this mangling; if the entity's context is enough to uniquely identify the entity, the simple `identifier` form is preferred.

Twenty operators of the form 'LA', 'LB', etc. are reserved to described entities related to the entity whose name is provided. For example, 'LE' and 'Le' in the "SC" module are used to represent the structs synthesized by the Clang importer for various "error code" enums.

Outlined bridged Objective C method call mangling includes which parameters and return value are bridged and the type of pattern outlined.

```
bridge-spec ::= bridged-kind bridged-param* bridged-return '_'

bridged-param ::= 'n' // not bridged parameter
bridged-param ::= 'b' // bridged parameter

bridged-return ::= 'n' // not bridged return
bridged-return ::= 'b' // bridged return

bridged-kind ::= 'm' // bridged method
bridged-kind ::= 'a' // bridged property (by address)
bridged-kind ::= 'p' // bridged property (by value)
```

Declaration Contexts

These manglings identify the enclosing context in which an entity was declared, such as its enclosing module, function, or nominal type.

```
context ::= module
context ::= entity
context ::= entity module generic-signature? 'E'
```

An `extension` mangling is used whenever an entity's declaration context is an extension *and* the entity being extended is in a different module. In this case the extension's module is mangled first, followed by the entity being extended. If the extension and the extended entity are in the same module, the plain `entity` mangling is preferred. If the extension is constrained, the constraints on the extension are mangled in its generic signature.

When mangling the context of a local entity within a constructor or destructor, the non-allocating or non-deallocating variant is used.

```
module ::= identifier                      // module name
module ::= known-module                    // abbreviation
```

```
context ::= entity identifier type-list 'XZ' // unknown runtime context
```

The runtime produces manglings of unknown runtime contexts when a declaration context has no preserved runtime information, or when a declaration is encoded in runtime in a way that the current runtime does not understand. These manglings are unstable and may change between runs of the process.

```
known-module ::= 's'                         // Swift
known-module ::= 'SC'                        // Clang-importer-synthesized
known-module ::= 'So'                        // C and Objective-C
```

The Objective-C module is used as the context for mangling Objective-C classes as `<type>`s.

Types

```
any-generic-type ::= substitution
any-generic-type ::= context decl-name 'C'     // nominal class type
any-generic-type ::= context decl-name 'O'     // nominal enum type
any-generic-type ::= context decl-name 'V'     // nominal struct type
any-generic-type ::= context decl-name 'XY'    // unknown nominal type
any-generic-type ::= protocol 'P'              // nominal protocol type

any-generic-type ::= standard-substitutions

standard-substitutions ::= 'S' KNOWN-TYPE-KIND       // known nominal type substitution
standard-substitutions ::= 'S' NATURAL KNOWN-TYPE-KIND    // repeated known type substitutions of the same kind

KNOWN-TYPE-KIND ::= 'A'                      // Swift.AutoreleasingUnsafeMutablePointer
KNOWN-TYPE-KIND ::= 'a'                      // Swift.Array
KNOWN-TYPE-KIND ::= 'B'                      // Swift.BinaryFloatingPoint
KNOWN-TYPE-KIND ::= 'b'                      // Swift.Bool
KNOWN-TYPE-KIND ::= 'c' KNOWN-TYPE-KIND-2    // Second set of standard types
KNOWN-TYPE-KIND ::= 'D'                      // Swift.Dictionary
KNOWN-TYPE-KIND ::= 'd'                      // Swift.Float64
KNOWN-TYPE-KIND ::= 'E'                      // Swift.Encodable
KNOWN-TYPE-KIND ::= 'e'                      // Swift.Decodable
KNOWN-TYPE-KIND ::= 'F'                      // Swift.FloatingPoint
KNOWN-TYPE-KIND ::= 'f'                      // Swift.Float32
KNOWN-TYPE-KIND ::= 'G'                      // Swift.RandomNumberGenerator
KNOWN-TYPE-KIND ::= 'H'                      // Swift.Hashable
KNOWN-TYPE-KIND ::= 'h'                      // Swift.Set
KNOWN-TYPE-KIND ::= 'I'                      // Swift.DefaultIndices
KNOWN-TYPE-KIND ::= 'i'                      // Swift.Int
KNOWN-TYPE-KIND ::= 'J'                      // Swift.Character
KNOWN-TYPE-KIND ::= 'j'                      // Swift.Numeric
KNOWN-TYPE-KIND ::= 'K'                      // Swift.BidirectionalCollection
KNOWN-TYPE-KIND ::= 'k'                      // Swift.RandomAccessCollection
KNOWN-TYPE-KIND ::= 'L'                      // Swift.Comparable
KNOWN-TYPE-KIND ::= 'l'                      // Swift.Collection
KNOWN-TYPE-KIND ::= 'M'                      // Swift.MutableCollection
KNOWN-TYPE-KIND ::= 'm'                      // Swift.RangeReplaceableCollection
KNOWN-TYPE-KIND ::= 'N'                      // Swift.ClosedRange
KNOWN-TYPE-KIND ::= 'n'                      // Swift.Range
KNOWN-TYPE-KIND ::= 'O'                      // Swift.ObjectIdentifier
KNOWN-TYPE-KIND ::= 'P'                      // Swift.UnsafePointer
KNOWN-TYPE-KIND ::= 'p'                      // Swift.UnsafeMutablePointer
KNOWN-TYPE-KIND ::= 'Q'                      // Swift.Equatable
KNOWN-TYPE-KIND ::= 'q'                      // Swift.Optional
KNOWN-TYPE-KIND ::= 'R'                      // Swift.UnsafeBufferPointer
KNOWN-TYPE-KIND ::= 'r'                      // Swift.UnsafeMutableBufferPointer
KNOWN-TYPE-KIND ::= 'S'                      // Swift.String
KNOWN-TYPE-KIND ::= 's'                      // Swift.Substring
KNOWN-TYPE-KIND ::= 'T'                      // Swift.Sequence
KNOWN-TYPE-KIND ::= 't'                      // Swift.IteratorProtocol
KNOWN-TYPE-KIND ::= 'U'                      // Swift.UnsignedInteger
KNOWN-TYPE-KIND ::= 'u'                      // Swift.UInt
KNOWN-TYPE-KIND ::= 'V'                      // Swift.UnsafeRawPointer
KNOWN-TYPE-KIND ::= 'v'                      // Swift.UnsafeMutableRawPointer
KNOWN-TYPE-KIND ::= 'W'                      // Swift.UnsafeRawBufferPointer
KNOWN-TYPE-KIND ::= 'w'                      // Swift.UnsafeMutableRawBufferPointer
KNOWN-TYPE-KIND ::= 'X'                      // Swift.RangeExpression
KNOWN-TYPE-KIND ::= 'x'                      // Swift.Strideable
KNOWN-TYPE-KIND ::= 'Y'                      // Swift.RawRepresentable
KNOWN-TYPE-KIND ::= 'y'                      // Swift.StringProtocol
KNOWN-TYPE-KIND ::= 'Z'                      // Swift.SignedInteger
KNOWN-TYPE-KIND ::= 'z'                      // Swift.BinaryInteger

KNOWN-TYPE-KIND-2 ::= 'A'        // Swift.Actor
KNOWN-TYPE-KIND-2 ::= 'C'        // Swift.CheckedContinuation
KNOWN-TYPE-KIND-2 ::= 'c'        // Swift.UnsafeContinuation
KNOWN-TYPE-KIND-2 ::= 'E'        // Swift.CancellationError
KNOWN-TYPE-KIND-2 ::= 'e'        // Swift.UnownedSerialExecutor
KNOWN-TYPE-KIND-2 ::= 'F'        // Swift.Executor
KNOWN-TYPE-KIND-2 ::= 'f'        // Swift.SerialExecutor
KNOWN-TYPE-KIND-2 ::= 'G'        // Swift.TaskGroup
KNOWN-TYPE-KIND-2 ::= 'g'        // Swift.ThrowingTaskGroup
KNOWN-TYPE-KIND-2 ::= 'I'        // Swift.AsyncIteratorProtocol
KNOWN-TYPE-KIND-2 ::= 'i'        // Swift.AsyncSequence
KNOWN-TYPE-KIND-2 ::= 'J'        // Swift.UnownedJob
KNOWN-TYPE-KIND-2 ::= 'M'        // Swift.MainActor
KNOWN-TYPE-KIND-2 ::= 'P'        // Swift.TaskPriority
KNOWN-TYPE-KIND-2 ::= 'S'        // Swift.AsyncStream
KNOWN-TYPE-KIND-2 ::= 's'        // Swift.AsyncThrowingStream
KNOWN-TYPE-KIND-2 ::= 'T'        // Swift.Task
KNOWN-TYPE-KIND-2 ::= 't'        // Swift.UnsafeCurrentTask

protocol ::= context decl-name
protocol ::= standard-substitutions

type ::= 'Bb'                                // Builtin.BridgeObject
type ::= 'BB'                                // Builtin.UnsafeValueBuffer
#if SWIFT_RUNTIME_VERSION >= 5.5
  type ::= 'Bc'                                // Builtin.RawUnsafeContinuation
  type ::= 'BD'                                // Builtin.DefaultActorStorage
  type ::= 'Be'                                // Builtin.Executor
#endif
type ::= 'Bf' NATURAL '_'                    // Builtin.Float<n>
type ::= 'Bi' NATURAL '_'                    // Builtin.Int<n>
type ::= 'BI'                                // Builtin.IntLiteral
#if SWIFT_RUNTIME_VERSION >= 5.5
  type ::= 'Bj'                                // Builtin.Job
#endif
type ::= 'BO'                                // Builtin.UnknownObject (no longer a distinct type, but still used for AnyObject)
type ::= 'Bo'                                // Builtin.NativeObject
type ::= 'Bp'                                // Builtin.RawPointer
type ::= 'Bt'                                // Builtin.SILToken
type ::= type 'Bv' NATURAL '_'               // Builtin.Vec<n>x<type>
type ::= 'Bw'                                // Builtin.Word
type ::= function-signature 'c'              // function type (escaping)
type ::= function-signature 'X' FUNCTION-KIND // special function type
type ::= bound-generic-type
type ::= type 'Sg'                           // optional type, shortcut for: type 'ySqG'
type ::= type 'Xo'                           // @unowned type
type ::= type 'Xu'                           // @unowned(unsafe) type
type ::= type 'Xw'                           // @weak type
type ::= impl-function-type 'XF'             // function implementation type (currently unused)
type ::= type 'Xb'                           // SIL @box type (deprecated)
type ::= type-list 'Xx'                      // SIL box type
type ::= type-list type-list generic-signature 'XX'
                                             // Generic SIL box type
type ::= type 'XD'                           // dynamic self type
```

```
type ::= type 'm'                          // metatype without representation
type ::= type 'XM' METATYPE-REPR           // metatype with representation
type ::= type 'Xp'                         // existential metatype without representation
type ::= type 'Xm' METATYPE-REPR           // existential metatype with representation
type ::= 'Xe'                              // error or unresolved type

bound-generic-type ::= type 'y' (type* '_')* type* retroactive-conformance* 'G'   // one type-list per nesting level of type
bound-generic-type ::= substitution

FUNCTION-KIND ::= 'f'                      // @thin function type
FUNCTION-KIND ::= 'U'                      // uncurried function type (currently not used)
FUNCTION-KIND ::= 'K'                      // @auto_closure function type (noescape)
FUNCTION-KIND ::= 'B'                      // objc block function type
FUNCTION-KIND ::= 'zB' C-TYPE              // objc block type with non-canonical C type
FUNCTION-KIND ::= 'L'                      // objc block function type with canonical C type (escaping) (DWARF only; otherwise use 'B' or
FUNCTION-KIND ::= 'C'                      // C function pointer / C++ method type
FUNCTION-KIND ::= 'zC' C-TYPE              // C function pointer / C++ method type with with non-canonical C type
FUNCTION-KIND ::= 'A'                      // @auto_closure function type (escaping)
FUNCTION-KIND ::= 'E'                      // function type (noescape)

C-TYPE is mangled according to the Itanium ABI, and prefixed with the length.
Non-ASCII identifiers are preserved as-is; we do not use Punycode.

function-signature ::= params-type params-type async? sendable? throws? differentiable? global-actor? // results and parameters

params-type ::= type 'z'? 'h'?             // tuple in case of multiple parameters or a single parameter with a single tuple type
                                           // with optional inout convention, shared convention. parameters don't have labels,
                                           // they are mangled separately as part of the entity.
params-type ::= empty-list                 // shortcut for no parameters

#if SWIFT_RUNTIME_VERSION >= 5.5
  async ::= 'Ya'                             // 'async' annotation on function types
  sendable ::= 'Yb'                          // @Sendable on function types
  global-actor :: = type 'Yc'                // Global actor on function type
#endif
throws ::= 'K'                             // 'throws' annotation on function types
differentiable ::= 'Yjf'                   // @differentiable(_forward) on function type
differentiable ::= 'Yjr'                   // @differentiable(reverse) on function type
differentiable ::= 'Yjd'                   // @differentiable on function type
differentiable ::= 'Yjl'                   // @differentiable(_linear) on function type

type-list ::= list-type '_' list-type*     // list of types
type-list ::= empty-list

                                                // FIXME: Consider replacing 'h' with a two-char code
list-type ::= type identifier? 'Yk'? 'z'? 'h'? 'n'? 'Yi'? 'd'? 'Yt'?  // type with optional label, '@noDerivative', inout convention, sha

METATYPE-REPR ::= 't'                      // Thin metatype representation
METATYPE-REPR ::= 'T'                      // Thick metatype representation
METATYPE-REPR ::= 'o'                      // ObjC metatype representation

type ::= archetype
type ::= associated-type
type ::= any-generic-type
type ::= protocol-list 'p'                 // existential type
type ::= protocol-list superclass 'Xc'     // existential type with superclass
type ::= protocol-list 'Xl'                // existential type with AnyObject
type ::= protocol-list 'y' (type* '_')* type* retroactive-conformance* 'Xp'   // parameterized protocol type
type ::= type-list 't'                     // tuple
type ::= type generic-signature 'u'        // generic type
type ::= 'x'                               // generic param, depth=0, idx=0
type ::= 'q' GENERIC-PARAM-INDEX           // dependent generic parameter
type ::= type assoc-type-name 'qa'         // associated type of non-generic param
type ::= assoc-type-name 'Qy' GENERIC-PARAM-INDEX  // associated type
type ::= assoc-type-name 'Qz'                      // shortcut for 'Qyz'
type ::= assoc-type-list 'QY' GENERIC-PARAM-INDEX  // associated type at depth
type ::= assoc-type-list 'QZ'                      // shortcut for 'QYz'

#if SWIFT_RUNTIME_VERSION >= 5.2
  type ::= type assoc-type-name 'Qx' // associated type relative to base `type`
  type ::= type assoc-type-list 'QX' // associated type relative to base `type`
#endif

protocol-list ::= protocol '_' protocol*
protocol-list ::= empty-list

assoc-type-list ::= assoc-type-name '_' assoc-type-name*

archetype ::= associated-type

associated-type ::= substitution
associated-type ::= protocol 'QP'          // self type of protocol
associated-type ::= archetype identifier 'Qa' // associated type

assoc-type-name ::= identifier                // associated type name without protocol
assoc-type-name ::= identifier protocol 'P'   //

empty-list ::= 'y'
```

Associated types use an abbreviated mangling when the base generic parameter or associated type is constrained by a single protocol requirement. The associated type in this case can be referenced unambiguously by name alone. If the base has multiple conformance constraints, then the protocol name is mangled in to disambiguate.

```
impl-function-type ::= type* 'I' FUNC-ATTRIBUTES '_'
impl-function-type ::= type* generic-signature 'I' FUNC-ATTRIBUTES '_'

FUNC-ATTRIBUTES ::= PATTERN-SUBS? INVOCATION-SUBS? PSEUDO-GENERIC? CALLEE-ESCAPE? DIFFERENTIABILITY-KIND? CALLEE-CONVENTION FUNC-REPRESEN

PATTERN-SUBS ::= 's'                       // has pattern substitutions
INVOCATION-SUB ::= 'I'                     // has invocation substitutions
PSEUDO-GENERIC ::= 'P'

CALLEE-ESCAPE ::= 'e'                      // @escaping (inverse of SIL @noescape)

DIFFERENTIABILITY-KIND ::= 'd'             // @differentiable
DIFFERENTIABILITY-KIND ::= 'l'             // @differentiable(_linear)
DIFFERENTIABILITY-KIND ::= 'f'             // @differentiable(_forward)
DIFFERENTIABILITY-KIND ::= 'r'             // @differentiable(reverse)

CALLEE-CONVENTION ::= 'y'                  // @callee_unowned
CALLEE-CONVENTION ::= 'g'                  // @callee_guaranteed
CALLEE-CONVENTION ::= 'x'                  // @callee_owned
CALLEE-CONVENTION ::= 't'                  // thin

FUNC-REPRESENTATION ::= 'B'                // C block invocation function
FUNC-REPRESENTATION ::= 'zB' C-TYPE        // C block invocation function with non-canonical C type
FUNC-REPRESENTATION ::= 'C'                // C global function
FUNC-REPRESENTATION ::= 'zC' C-TYPE        // C global function with non-canonical C type
FUNC-REPRESENTATION ::= 'M'                // Swift method
FUNC-REPRESENTATION ::= 'J'                // ObjC method
FUNC-REPRESENTATION ::= 'K'                // closure
FUNC-REPRESENTATION ::= 'W'                // protocol witness

COROUTINE-KIND ::= 'A'                     // yield-once coroutine
COROUTINE-KIND ::= 'G'                     // yield-many coroutine

#if SWIFT_RUNTIME_VERSION >= 5.5
  SENDABLE ::= 'h'                           // @Sendable
  ASYNC ::= 'H'                              // @async
#endif
```

```
PARAM-CONVENTION ::= 'i'                    // indirect in
PARAM-CONVENTION ::= 'c'                    // indirect in constant
PARAM-CONVENTION ::= 'l'                    // indirect inout
PARAM-CONVENTION ::= 'b'                    // indirect inout aliasable
PARAM-CONVENTION ::= 'n'                    // indirect in guaranteed
PARAM-CONVENTION ::= 'x'                    // direct owned
PARAM-CONVENTION ::= 'y'                    // direct unowned
PARAM-CONVENTION ::= 'g'                    // direct guaranteed
PARAM-CONVENTION ::= 'e'                    // direct deallocating

PARAM-DIFFERENTIABILITY ::= 'w'             // @noDerivative

RESULT-CONVENTION ::= 'r'                   // indirect
RESULT-CONVENTION ::= 'o'                   // owned
RESULT-CONVENTION ::= 'd'                   // unowned
RESULT-CONVENTION ::= 'u'                   // unowned inner pointer
RESULT-CONVENTION ::= 'a'                   // auto-released

RESULT-DIFFERENTIABILITY ::= 'w'             // @noDerivative
```

For the most part, manglings follow the structure of formal language types. However, in some cases it is more useful to encode the exact implementation details of a function type.

```
#if SWIFT_VERSION >= 5.1
  type ::= 'Qr'                         // opaque result type (of current decl, used for the first opaque type parameter only)
  type ::= 'QR' INDEX                   // same as above, for subsequent opaque type parameters, INDEX is the ordinal -1
  type ::= opaque-type-decl-name bound-generic-args 'Qo' INDEX // opaque type

  opaque-type-decl-name ::= entity 'QO' // opaque result type of specified decl
#endif

#if SWIFT_VERSION >= 5.4
  type ::= 'Qu'                         // opaque result type (of current decl, first param)
                                        // used for ObjC class runtime name purposes.
  type ::= 'QU' INDEX
#endif
```

Opaque return types have a special short representation in the mangling of their defining entity. In structural position, opaque types are fully qualified by mangling the defining entity for the opaque declaration and the substitutions into the defining entity's generic environment.

The `type*` list contains parameter and return types (including the error result), in that order. The number of parameters and results must match with the number of `<PARAM-CONVENTION>` and `<RESULT-CONVENTION>` characters after the `<FUNC-REPRESENTATION>`. The `<generic-signature>` is used if the function is polymorphic.

DWARF debug info and USRs also mangle sugared types, adding the following productions:

```
any-generic-type ::= context decl-name 'a'     // typealias type
type ::= base-type "XSq"                       // sugared Optional type
type ::= base-type "XSa"                       // sugared Array type
type ::= key-type value-type "XSD"             // sugared Dictionary type
type ::= base-type "XSp"                       // sugared Paren type
```

Generics

```
protocol-conformance-context ::= protocol module generic-signature?

protocol-conformance ::= type protocol-conformance-context
```

`<protocol-conformance>` refers to a type's conformance to a protocol. The named module is the one containing the extension or type declaration that declared the conformance.

```
protocol-conformance ::= type protocol
```

If `type` is a generic parameter or associated type of one, then no module is mangled, because the conformance must be resolved from the generic environment.

protocol-conformance ::= context identifier protocol identifier generic-signature? // Property behavior conformance

Property behaviors are implemented using private protocol conformances.

```
concrete-protocol-conformance ::= type protocol-conformance-ref any-protocol-conformance-list 'HC'
protocol-conformance-ref ::= protocol 'HP'   // same module as conforming type
protocol-conformance-ref ::= protocol 'Hp'   // same module as protocol
protocol-conformance-ref ::= protocol module // "retroactive"

any-protocol-conformance ::= concrete-protocol-conformance
any-protocol-conformance ::= dependent-protocol-conformance

any-protocol-conformance-list ::= any-protocol-conformance '_' any-protocol-conformance-list
any-protocol-conformance-list ::= empty-list

DEPENDENT-CONFORMANCE-INDEX ::= INDEX

dependent-protocol-conformance ::= type protocol 'HD' DEPENDENT-CONFORMANCE-INDEX
dependent-protocol-conformance ::= dependent-protocol-conformance protocol 'HI' DEPENDENT-CONFORMANCE-INDEX
dependent-protocol-conformance ::= dependent-protocol-conformance
    dependent-associated-conformance 'HA' DEPENDENT-CONFORMANCE-INDEX

dependent-associated-conformance ::= type protocol
dependent-protocol-conformance ::= dependent-protocol-conformance opaque-type 'HO'
```

A compact representation used to represent mangled protocol conformance witness arguments at runtime. The `module` is only specified for conformances that are "retroactive", meaning that the context in which the conformance is defined is in neither the protocol or type module. For a non-retroactive conformance where both the type *and* the protocol are in the same module, or for synthesized conformances that have no owning module, the "HP" operator is preferred. The concrete protocol conformances that follow are for the conditional conformance requirements.

Dependent protocol conformances mangle the access path required to extract a protocol conformance from some conformance passed into the environment. The first case (operator "HD") is the leaf requirement, containing a dependent type and the protocol it conforms to. The remaining dependent protocol conformance manglings describe lookups performed on their child dependent protocol conformances. The "HI" operator retrieves the named inherited protocol from the witness table produced by the child. The "HA" operator refers to an associated conformance within the witness table, identified by the dependent type and protocol. In all cases, the DEPENDENT-CONFORMANCE-INDEX is an INDEX value indicating the position of the appropriate value within the generic environment (for "HD") or witness table (for "HI" and "HA") when it is known to be at a fixed position. An index of 1 ("0_") is used to indicate "unknown"; all other values are adjusted by 2. That these indexes are not 0-based is a bug that's now codified into the ABI; the index 0 is therefore reserved.

```
generic-signature ::= requirement* 'l'     // one generic parameter
generic-signature ::= requirement* 'r' GENERIC-PARAM-COUNT* 'l'

GENERIC-PARAM-COUNT ::= 'z'                 // zero parameters
GENERIC-PARAM-COUNT ::= INDEX               // N+1 parameters

requirement ::= protocol 'R' GENERIC-PARAM-INDEX                  // protocol requirement
requirement ::= protocol assoc-type-name 'Rp' GENERIC-PARAM-INDEX // protocol requirement on associated type
requirement ::= protocol assoc-type-list 'RP' GENERIC-PARAM-INDEX // protocol requirement on associated type at depth
requirement ::= protocol substitution 'RQ'                        // protocol requirement with substitution
requirement ::= type 'Rb' GENERIC-PARAM-INDEX                     // base class requirement
requirement ::= type assoc-type-name 'Rc' GENERIC-PARAM-INDEX     // base class requirement on associated type
requirement ::= type assoc-type-list 'RC' GENERIC-PARAM-INDEX     // base class requirement on associated type at depth
requirement ::= type substitution 'RB'                            // base class requirement with substitution
requirement ::= type 'Rs' GENERIC-PARAM-INDEX                     // same-type requirement
requirement ::= type assoc-type-name 'Rt' GENERIC-PARAM-INDEX     // same-type requirement on associated type
requirement ::= type assoc-type-list 'RT' GENERIC-PARAM-INDEX     // same-type requirement on associated type at depth
```

```
requirement ::= type substitution 'RS'                             // same-type requirement with substitution
requirement ::= type 'Rl' GENERIC-PARAM-INDEX LAYOUT-CONSTRAINT   // layout requirement
requirement ::= type assoc-type-name 'Rm' GENERIC-PARAM-INDEX LAYOUT-CONSTRAINT    // layout requirement on associated type
requirement ::= type assoc-type-list 'RM' GENERIC-PARAM-INDEX LAYOUT-CONSTRAINT    // layout requirement on associated type at depth
requirement ::= type substitution 'RM' LAYOUT-CONSTRAINT                           // layout requirement with substitution

GENERIC-PARAM-INDEX ::= 'z'                // depth = 0,   idx = 0
GENERIC-PARAM-INDEX ::= INDEX              // depth = 0,   idx = N+1
GENERIC-PARAM-INDEX ::= 'd' INDEX INDEX    // depth = M+1, idx = N

LAYOUT-CONSTRAINT ::= 'N'  // NativeRefCountedObject
LAYOUT-CONSTRAINT ::= 'R'  // RefCountedObject
LAYOUT-CONSTRAINT ::= 'T'  // Trivial
LAYOUT-CONSTRAINT ::= 'C'  // Class
LAYOUT-CONSTRAINT ::= 'D'  // NativeClass
LAYOUT-CONSTRAINT ::= 'E' LAYOUT-SIZE-AND-ALIGNMENT  // Trivial of exact size
LAYOUT-CONSTRAINT ::= 'e' LAYOUT-SIZE  // Trivial of exact size
LAYOUT-CONSTRAINT ::= 'M' LAYOUT-SIZE-AND-ALIGNMENT  // Trivial of size at most N bits
LAYOUT-CONSTRAINT ::= 'm' LAYOUT-SIZE  // Trivial of size at most N bits
LAYOUT-CONSTRAINT ::= 'U'  // Unknown layout

LAYOUT-SIZE ::= INDEX // Size only
LAYOUT-SIZE-AND-ALIGNMENT ::= INDEX INDEX // Size followed by alignment
```

A generic signature begins with an optional list of requirements. The `<GENERIC-PARAM-COUNT>` describes the number of generic parameters at each depth of the signature. As a special case, no `<GENERIC-PARAM-COUNT>` values indicates a single generic parameter at the outermost depth:

```
x_xCru                           // <T_0_0> T_0_0 -> T_0_0
d_0__xCr_0_u                     // <T_0_0><T_1_0, T_1_1> T_0_0 -> T_1_1
```

A generic signature must only precede an operator character which is different from any character in a `<GENERIC-PARAM-COUNT>`.

```
retroactive-conformance ::= any-protocol-conformance 'g' INDEX
```

When a protocol conformance used to satisfy one of a bound generic type's generic requirements is retroactive (i.e., it is specified in a module other than the module of the conforming type or the conformed-to protocol), it is mangled with its offset into the set of conformance requirements, the root protocol conformance, and the suffix 'g'.

Identifiers

```
identifier ::= substitution
identifier ::= NATURAL IDENTIFIER-STRING   // identifier without word substitutions
identifier ::= '0' IDENTIFIER-PART         // identifier with word substitutions

IDENTIFIER-PART ::= NATURAL IDENTIFIER-STRING
IDENTIFIER-PART ::= [a-z]                  // word substitution (except the last one)
IDENTIFIER-PART ::= [A-Z]                  // last word substitution in identifier

IDENTIFIER-STRING ::= IDENTIFIER-START-CHAR IDENTIFIER-CHAR*
IDENTIFIER-START-CHAR ::= [_a-zA-Z]
IDENTIFIER-CHAR ::= [_$a-zA-Z0-9]
```

`<identifier>` is run-length encoded: the natural indicates how many characters follow. Operator characters are mapped to letter characters as given. In neither case can an identifier start with a digit, so there's no ambiguity with the run-length.

If the run-length start with a `0` the identifier string contains word substitutions. A word is a sub-string of an identifier which contains letters and digits `[A-Za-z0-9]`. Words are separated by underscores _. In addition a new word begins with an uppercase letter `[A-Z]` if the previous character is not an uppercase letter:

```
Abc1DefG2HI          // contains four words 'Abc1', 'Def' and 'G2' and 'HI'
_abc1_def_G2hi       // contains three words 'abc1', 'def' and G2hi
```

The words of all identifiers, which are encoded in the current mangling are enumerated and assigned to a letter: a = first word, b = second word, etc.

An identifier containing word substitutions is a sequence of run-length encoded sub-strings and references to previously mangled words. All but the last word-references are lowercase letters and the last one is an uppercase letter. If there is no literal sub-string after the last word-reference, the last word-reference is followed by a `0`.

Let's assume the current mangling already encoded the identifier `AbcDefGHI`:

```
02Myac1_B    // expands to: MyAbcGHI_Def
```

A maximum of 26 words in a mangling can be used for substitutions.

```
identifier ::= '00' natural '_'? IDENTIFIER-CHAR+  // '_' is inserted if the identifier starts with a digit or '_'.
```

Identifiers that contain non-ASCII characters are encoded using the Punycode algorithm specified in RFC 3492, with the modifications that _ is used as the encoding delimiter, and uppercase letters A through J are used in place of digits 0 through 9 in the encoding character set. The mangling then consists of an `00` followed by the run length of the encoded string and the encoded string itself. For example, the identifier `vergÃ¼enza` is mangled to `0012vergenza_JFa`. (The encoding in standard Punycode would be `vergenza-95a`)

If the encoded string starts with a digit or an _, an additional _ is inserted between the run length and the encoded string.

```
identifier ::= identifier 'o' OPERATOR-FIXITY

OPERATOR-FIXITY ::= 'p'                    // prefix operator
OPERATOR-FIXITY ::= 'P'                    // postfix operator
OPERATOR-FIXITY ::= 'i'                    // infix operator

OPERATOR-CHAR ::= 'a'                      // & 'and'
OPERATOR-CHAR ::= 'c'                      // @ 'commercial at'
OPERATOR-CHAR ::= 'd'                      // / 'divide'
OPERATOR-CHAR ::= 'e'                      // = 'equals'
OPERATOR-CHAR ::= 'g'                      // > 'greater'
OPERATOR-CHAR ::= 'l'                      // < 'less'
OPERATOR-CHAR ::= 'm'                      // * 'multiply'
OPERATOR-CHAR ::= 'n'                      // ! 'not'
OPERATOR-CHAR ::= 'o'                      // | 'or'
OPERATOR-CHAR ::= 'p'                      // + 'plus'
OPERATOR-CHAR ::= 'q'                      // ? 'question'
OPERATOR-CHAR ::= 'r'                      // % 'remainder'
OPERATOR-CHAR ::= 's'                      // - 'subtract'
OPERATOR-CHAR ::= 't'                      // ~ 'tilde'
OPERATOR-CHAR ::= 'x'                      // ^ 'xor'
OPERATOR-CHAR ::= 'z'                      // . 'zperiod'
```

If an identifier is followed by an `o` its text is interpreted as an operator. Each lowercase character maps to an operator character (`OPERATOR-CHAR`).

Operators that contain non-ASCII characters are mangled by first mapping the ASCII operator characters to letters as for pure ASCII operator names, then Punycode-encoding the substituted string. For example, the infix operator `Â«+Â»` is mangled to `007p_qcaDcoi` (`p_qcaDc` being the encoding of the substituted string `Â«pÂ»`).

Substitutions

```
substitution ::= 'A' INDEX                  // substitution of N+26
substitution ::= 'A' SUBST_IDX* LAST-SUBST-IDX    // One or more consecutive substitutions of N < 26
SUBST-IDX ::= [a-z]
SUBST-IDX ::= NATURAL [a-z]
LAST-SUBST-IDX ::= [A-Z]
LAST-SUBST-IDX ::= NATURAL [A-Z]
```

`<substitution>` is a back-reference to a previously mangled entity. The mangling algorithm maintains a mapping of entities to substitution indices as it runs. When an entity that can be represented by a substitution (a module, nominal type, or protocol) is mangled, a substitution is first looked for in the substitution map, and if it is present, the entity is mangled using the associated substitution index. Otherwise, the entity is mangled normally, and it is then added to the substitution map and associated with the next available substitution index.

For example, in mangling a function type `(zim.zang.zung, zim.zang.zung, zim.zippity) -> zim.zang.zoo` (with module `zim` and class `zim.zang`), the recurring contexts `zim`, `zim.zang`, and `zim.zang.zung` will be mangled using substitutions after being mangled for the first time. The first argument type will mangle in long form, `3zim4zang4zung`, and in doing so, `zim` will acquire substitution `AA`, `zim.zang` will acquire substitution `AB`, and `zim.zang.zung` will acquire `AC`. The second argument is the same as the first and will mangle using its substitution, `AC`. The third argument type will mangle using the substitution for `zim`, `AA7zippity`. (It also acquires substitution `AD` which would be used if it mangled again.) The result type will mangle using the substitution for `zim.zang`, `AB3zoo` (and acquire substitution `AE`).

There are some pre-defined substitutions, see `KNOWN-TYPE-KIND`.

If the mangling contains two or more consecutive substitutions, it can be abbreviated with the `A` substitution. Similar to word-substitutions the index is encoded as letters, whereas the last letter is uppercase:

```
AaeB      // equivalent to A_A4_A0_
```

Repeated substitutions are encoded with a natural prefix number:

```
A3a2B     // equivalent to AaaabB
```

Numbers and Indexes

```
INDEX ::= '_'                               // 0
INDEX ::= NATURAL '_'                       // N+1
NATURAL ::= [1-9] [0-9]*
NATURAL_ZERO ::= [0-9]+
```

`<INDEX>` is a production for encoding numbers in contexts that can't end in a digit; it's optimized for encoding smaller numbers.

```
INDEX-SUBSET ::= ('S' | 'U')+
```

`<INDEX-SUBSET>` is encoded like a bit vector and is optimized for encoding indices with a small upper bound.

Function Specializations

```
specialization ::= type '_' type* 'Tg' SPEC-INFO     // Generic re-abstracted specialization
specialization ::= type '_' type* 'TB' SPEC-INFO     // Alternative mangling for generic re-abstracted specializations,
                                                     // used for functions with re-abstracted resilient parameter types.
specialization ::= type '_' type* 'Ts' SPEC-INFO     // Generic re-abstracted prespecialization
specialization ::= type '_' type* 'TG' SPEC-INFO     // Generic not re-abstracted specialization
specialization ::= type '_' type* 'Ti' SPEC-INFO     // Inlined function with generic substitutions.
```

The types are the replacement types of the substitution list.

```
specialization ::= type 'Tp' SPEC-INFO // Partial generic specialization
specialization ::= type 'TP' SPEC-INFO // Partial generic specialization, not re-abstracted
```

The type is the function type of the specialized function.

```
specialization ::= spec-arg* 'Tf' SPEC-INFO ARG-SPEC-KIND* '_' ARG-SPEC-KIND  // Function signature specialization kind
```

The `<ARG-SPEC-KIND>` describes how arguments are specialized. Some kinds need arguments, which precede `Tf`.

```
spec-arg ::= identifier
spec-arg ::= type

SPEC-INFO ::= FRAGILE? PASSID

PASSID ::= '0'                             // AllocBoxToStack,
PASSID ::= '1'                             // ClosureSpecializer,
PASSID ::= '2'                             // CapturePromotion,
PASSID ::= '3'                             // CapturePropagation,
PASSID ::= '4'                             // FunctionSignatureOpts,
PASSID ::= '5'                             // GenericSpecializer,

FRAGILE ::= 'q'

ARG-SPEC-KIND ::= 'n'                      // Unmodified argument
ARG-SPEC-KIND ::= 'c'                      // Consumes n 'type' arguments which are closed over types in argument order
                                           // and one 'identifier' argument which is the closure symbol name
ARG-SPEC-KIND ::= 'p' CONST-PROP           // Constant propagated argument
ARG-SPEC-KIND ::= 'e' 'D'? 'G'? 'X'?       // Generic argument, with optional dead, owned=>guaranteed or exploded-specifier
ARG-SPEC-KIND ::= 'd' 'G'? 'X'?            // Dead argument, with optional owned=>guaranteed or exploded-specifier
ARG-SPEC-KIND ::= 'g' 'X'?                 // Owned => Guaranteed,, with optional exploded-specifier
ARG-SPEC-KIND ::= 'x'                      // Exploded
ARG-SPEC-KIND ::= 'i'                      // Box to value
ARG-SPEC-KIND ::= 's'                      // Box to stack

CONST-PROP ::= 'f'                         // Consumes one identifier argument which is a function symbol name
CONST-PROP ::= 'g'                         // Consumes one identifier argument which is a global symbol name
CONST-PROP ::= 'i' NATURAL_ZERO            // 64-bit-integer
CONST-PROP ::= 'd' NATURAL_ZERO            // float-as-64-bit-integer
CONST-PROP ::= 's' ENCODING                // string literal. Consumes one identifier argument.
CONST-PROP ::= 'k'                         // keypath. Consumes one identifier - the SHA1 of the keypath and two types (root and value).

ENCODING ::= 'b'                           // utf8
ENCODING ::= 'w'                           // utf16
ENCODING ::= 'c'                           // utf16
```

If the first character of the string literal is a digit `[0-9]` or an underscore _, the identifier for the string literal is prefixed with an additional underscore _.

Conventions for foreign symbols

Swift interoperates with multiple other languages - C, C++, Objective-C, and Objective-C++. Each of these languages defines their own mangling conventions, so Swift must take care to follow them. However, these conventions do not cover Swift-specific symbols like Swift type metadata for foreign types, so Swift uses its own mangling scheme for those symbols.

Importing C and C++ structs

Types imported from C and C++ are imported as if they are located in the `__C` module, regardless of the actual Clang module that they are coming from. This can be observed when mangling a Swift function that accepts a C/C++ struct as a parameter:

C++ module `CxxStructModule`:

```
struct CxxStruct {};

inline void cxxFunction(CxxStruct s) {}
```

Swift module `main` that imports `CxxStructModule`:

```
import CxxStructModule

public func swiftFunction(_ s: CxxStruct) {}
```

Resulting symbols (showing only Itanium-mangled C++ symbols for brevity):

```
_Z11cxxFunction9CxxStruct // -> cxxFunction(CxxStruct)
s4main13swiftFunctionyySo9CxxStructVF // -> main.swiftFunction(__C.CxxStruct) -> ()
```

The reason for ignoring the Clang module and always putting C and C++ types into `__C` at the Swift ABI level is that the Clang module is not a part of the C or C++ ABI. When owners of C and C++ Clang modules decide what changes are ABI-compatible or not, they will likely take into account C and C++ ABI, but not the Swift ABI. Therefore, Swift ABI can only encode information about a C or C++ type that the C and C++ ABI already encodes in order to remain compatible with future versions of libraries that evolve according to C and C++ ABI compatibility principles.

The C/C++ compiler does not generate Swift metadata symbols and value witness tables for C and C++ types. To make a foreign type usable in Swift in the same way as a native type, the Swift compiler must generate these symbols. Specifically, each Swift module that uses a given C or C++ type generates the necessary Swift symbols. For the example above the Swift compiler will generate following nominal type descriptor symbol for `CxxStruct` while compiling the `main` module:

```
sSo9CxxStructVMn // -> nominal type descriptor for __C.CxxStruct
```

Importing C++ class template instantiations

A class template instantiation is imported as a struct named `__CxxTemplateInst` plus Itanium mangled type of the instantiation (see the `type` production in the Itanium specification). Note that Itanium mangling is used on all platforms, regardless of the ABI of the C++ toolchain, to ensure that the mangled name is a valid Swift type name (this is not the case for MSVC mangled names). A prefix with a double underscore (to ensure we have a reserved C++ identifier) is added to limit the possibility for conflicts with names of user-defined structs. The struct is notionally defined in the `__C` module, similarly to regular C and C++ structs and classes. Consider the following C++ module:

```
template<class T>
struct MagicWrapper {
  T t;
};

struct MagicNumber {};

typedef MagicWrapper<MagicNumber> WrappedMagicNumber;
```

`WrappedMagicNumber` is imported as a typealias for struct `__CxxTemplateInst12MagicWrapperI11MagicNumberE`. Interface of the imported module looks as follows:

```
struct __CxxTemplateInst12MagicWrapperI11MagicNumberE {
  var t: MagicNumber
}
struct MagicNumber {}
typealias WrappedMagicNumber = __CxxTemplateInst12MagicWrapperI11MagicNumberE
```