orphan:

Swift Language Reference Manual

Contents

Introduction

> **Warning**
>
> This document has not been kept up to date.

> **Commentary**
>
> In addition to the main spec, there are lots of open ended questions, justification, and ideas of what best practices should be. That random discussion is placed in boxes like this one to clarify what is normative and what is discussion.

This is the language reference manual for the Swift language, which is highly volatile and constantly under development. As the prototype evolves, this document should be kept up to date with what is actually implemented.

The grammar and structure of the language is defined in BNF form in yellow boxes. Examples are shown in gray boxes, and assume that the standard library is in use (unless otherwise specified).

Basic Goals

> **Commentary**

A non-goal of the Swift project in general is to become some amazing research project. We really want to focus on delivering a real product, and having the design and spec co-evolve.

In no particular order, and not explained well:

- Support building great frameworks and applications, with a specific focus on permitting rich and powerful APIs.
- Get the defaults right: this reduces the barrier to entry and increases the odds that the right thing happens.
- Through our support for building great APIs, we aim to provide an expressive and productive language that is fun to program in.
- Support low-level system programming. We should want to write compilers, operating system kernels, and media codecs in Swift. This means that being able to obtain high performance is really quite important.
- Provide really great tools, like an IDE, debugger, profiling, etc.
- Where possible, steal great ideas instead of innovating new things that will work out in unpredictable ways. It turns out that there are a lot of good ideas already out there.
- Memory safe by default: array overrun errors, uninitialized values, and other problems endemic to C should not occur in Swift, even if it means some amount of runtime overhead. Eventually these checks will be disableable for people who want ultimate performance in production builds.
- Efficiently implementable with a static compiler: runtime compilation is great technology and Swift may eventually get a runtime optimizer, but it is a strong goal to be able to implement swift with just a static compiler.
- Interoperate as transparently as possible with C, Objective-C, and C++ without having to write an equivalent of "extern C" for every referenced definition.
- Great support for efficient by-value types.
- Elegant and natural syntax, aiming to be familiar and easy to transition to for "C" people. Differences from the C family should only be done when it provides a significant win (e.g. eliminate declarator syntax).
- Lots of other stuff too.

A smaller wishlist goal is to support embedded sub-languages in swift, so that we don't get the OpenCL-is-like-C-but-very-different-in-many-details problem.

Basic Approach

Commentary

Pushing as much of the language as realistic out of the compiler and into the library is generally good for a few reasons: 1) we end up with a smaller core language. 2) we force the language that is left to be highly expressive and extensible. 3) this highly expressive language core can then be used to build a lot of other great libraries, hopefully many we can't even anticipate at this point.

The basic approach in designing and implementing the Swift prototype was to start at the very bottom of the stack (simple expressions and the trivial bits of the type system) and incrementally build things up one brick at a time. There is a big focus on making things as simple as possible and having a clean internal core. Where it makes sense, sugar is added on top to make the core more expressive for common situations.

One major aspect that dovetails with expressivity, learnability, and focus on API development is that much of the language is implemented in a :ref:`standard library <langref.stdlib>` (inspired in part by the Haskell Standard Prelude). This means that things like `Int` and `Void` are not part of the language itself, but are instead part of the standard library.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst`, line 92); *backlink*

Unknown interpreted text role "ref".

Phases of Translation

Commentary

Because Swift doesn't rely on a C-style "lexer hack" to know what is a type and what is a value, it is possible to fully parse a file without resolving import declarations.

Swift has a strict separation between its phases of translation, and the compiler follows a conceptually simple design. The phases of translation are:

- :ref:`Lexing <langref.lexical>`: A source file is broken into tokens according to a (nearly, `/**/` comments can be nested) regular grammar.

  System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst`, line 111); *backlink*

  Unknown interpreted text role "ref".

- Parsing and AST Building: The tokens are parsed according to the grammar set out below. The grammar is context free and does not require any "type feedback" from the lexer or later stages. During parsing, name binding for references to local variables and other declarations that are not at module (and eventually namespace) scope are bound.
- :ref:`Name Binding <langref.namebind>`: At this phase, references to non-local types and values are bound, and :ref:`import directives <langref.decl.import>` are both validated and searched. Name binding can cause recursive compilation of modules that are referenced but not yet built.

  System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst`, line 120); *backlink*

  Unknown interpreted text role "ref".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 120); *backlink*

Unknown interpreted text role "ref".

- :ref:`Type Checking <langref.typecheck>`: During this phase all types are resolved within value definitions, :ref:`function application <langref.expr.call>` and binary expressions are found and formed, and overloaded functions are resolved.

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 125); *backlink*

  Unknown interpreted text role "ref".

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 125); *backlink*

  Unknown interpreted text role "ref".

- Code Generation: The AST is converted the LLVM IR, optimizations are performed, and machine code generated.
- Linking: runtime libraries and referenced modules are linked in.

FIXME: "import Swift" implicitly added as the last import in a source file.

Lexical Structure

> **Commentary**
>
> Not all characters are "taken" in the language, this is because it is still growing. As there becomes a reason to assign things into the identifier or punctuation bucket, we will do so as swift evolves.

The lexical structure of a Swift file is very simple: the files are tokenized according to the following productions and categories. As is usual with most languages, tokenization uses the maximal munch rule and whitespace separates tokens. This means that "`a b`" and "`ab`" lex into different token streams and are therefore different in the grammar.

Whitespace and Comments

> **Commentary**
>
> Nested block comments are important because we don't have the nestable `#if 0` hack from C to rely on.

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 164)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

  whitespace ::= ' '
  whitespace ::= '\n'
  whitespace ::= '\r'
  whitespace ::= '\t'
  whitespace ::= '\0'
  comment    ::= //.*[\n\r]
  comment    ::= /* .... */
```

Space, newline, tab, and the nul byte are all considered whitespace and are discarded, with one exception: a `'(' or '[` which does not follow a non-whitespace character is different kind of token (called *spaced*) from one which does not (called *unspaced*). A `'(' or '[` at the beginning of a file is spaced.

Comments may follow the BCPL style, starting with a "`//`" and running to the end of the line, or may be recursively nested `/**/` style comments. Comments are ignored and treated as whitespace.

Reserved Punctuation Tokens

> **Commentary**
>
> Note that `->` is used for function types `() -> Int`, not pointer dereferencing.

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 194)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

  punctuation ::= '('
  punctuation ::= ')'
  punctuation ::= '{'
  punctuation ::= '}'
  punctuation ::= '['
  punctuation ::= ']'
  punctuation ::= '.'
  punctuation ::= ','
```

```
punctuation ::= ';'
punctuation ::= ':'
punctuation ::= '='
punctuation ::= '->'
punctuation ::= '&' // unary prefix operator
```

These are all reserved punctuation that are lexed into tokens. Most other non-alphanumeric characters are matched as :ref:`operators <langref.lexical.operator>`. Unlike operators, these tokens are not overloadable.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 210); *backlink*

Unknown interpreted text role "ref".

Reserved Keywords

Commentary

The number of keywords is reduced by pushing most functionality into the library (e.g. "builtin" datatypes like `Int` and `Bool`). This allows us to add new stuff to the library in the future without worrying about conflicting with the user's namespace.

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 227)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

  // Declarations and Type Keywords
  keyword ::= 'class'
  keyword ::= 'destructor'
  keyword ::= 'extension'
  keyword ::= 'import'
  keyword ::= 'init'
  keyword ::= 'func'
  keyword ::= 'enum'
  keyword ::= 'protocol'
  keyword ::= 'struct'
  keyword ::= 'subscript'
  keyword ::= 'Type'
  keyword ::= 'typealias'
  keyword ::= 'var'
  keyword ::= 'where'

  // Statements
  keyword ::= 'break'
  keyword ::= 'case'
  keyword ::= 'continue'
  keyword ::= 'default'
  keyword ::= 'do'
  keyword ::= 'else'
  keyword ::= 'if'
  keyword ::= 'in'
  keyword ::= 'for'
  keyword ::= 'return'
  keyword ::= 'switch'
  keyword ::= 'then'
  keyword ::= 'while'

  // Expressions
  keyword ::= 'as'
  keyword ::= 'is'
  keyword ::= 'new'
  keyword ::= 'super'
  keyword ::= 'self'
  keyword ::= 'Self'
  keyword ::= 'type'
  keyword ::= '__COLUMN__'
  keyword ::= '__FILE__'
  keyword ::= '__LINE__'
```

These are the builtin keywords. Keywords can still be used as names via *escaped identifiers <langref.lexical.escapedident>*.

Contextual Keywords

Swift uses several contextual keywords at various parts of the language. Contextual keywords are not reserved words, meaning that they can be used as identifiers. However, in certain contexts, they act as keywords, and are represented as such in the grammar below. The following identifiers act as contextual keywords within the language:

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 285)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

  get
  infix
  mutating
  nonmutating
  operator
  override
```

```
postfix
prefix
set
```

Integer Literals

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 302)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

  integer_literal ::= [0-9][0-9_]*
  integer_literal ::= 0x[0-9a-fA-F][0-9a-fA-F_]*
  integer_literal ::= 0o[0-7][0-7_]*
  integer_literal ::= 0b[01][01_]*
```

Integer literal tokens represent simple integer values of unspecified precision. They may be expressed in decimal, binary with the '`0b`' prefix, octal with the '`0o`' prefix, or hexadecimal with the '`0x`' prefix. Unlike C, a leading zero does not affect the base of the literal.

Integer literals may contain underscores at arbitrary positions after the first digit. These underscores may be used for human readability and do not affect the value of the literal.

```
789
0789

1000000
1_000_000

0b111_101_101
0o755

0b1111_1011
0xFB
```

Floating Point Literals

Commentary

We require a digit on both sides of the dot to allow lexing "`4.km`" as "`4 . km`" instead of "`4. km`" and for a series of dots to be an operator (for ranges). The regex for decimal literals is same as Java, and the one for hex literals is the same as C99, except that we do not allow a trailing suffix that specifies a precision.

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 345)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

  floating_literal ::= [0-9][0-9_]*\.[0-9][0-9_]*
  floating_literal ::= [0-9][0-9_]*\.[0-9][0-9_]*[eE][+-]?[0-9][0-9_]*
  floating_literal ::= [0-9][0-9_]*[eE][+-]?[0-9][0-9_]*
  floating_literal ::= 0x[0-9A-Fa-f][0-9A-Fa-f_]*
                       (\.[0-9A-Fa-f][0-9A-Fa-f_]*)?[pP][+-]?[0-9][0-9_]*
```

Floating point literal tokens represent floating point values of unspecified precision. Decimal and hexadecimal floating-point literals are supported.

The integer, fraction, and exponent of a floating point literal may each contain underscores at arbitrary positions after their first digits. These underscores may be used for human readability and do not affect the value of the literal. Each part of the floating point literal must however start with a digit; `1._0` would be a reference to the `_0` member of `1`.

```
1.0
1000000.75
1_000_000.75

0x1.FFFFFFFFFFFFFp1022
0x1.FFFF_FFFF_FFFF_Fp1_022
```

Character Literals

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 376)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

  character_literal ::= '[^'\\\n\r]|character_escape'
  character_escape  ::= [\]0 [\][\] | [\]t | [\]n | [\]r | [\]" | [\]'
  character_escape  ::= [\]x hex hex
  character_escape  ::= [\]u hex hex hex hex
  character_escape  ::= [\]U hex hex hex hex hex hex hex hex
  hex               ::= [0-9a-fA-F]
```

`character_literal` tokens represent a single character, and are surrounded by single quotes.

The ASCII and Unicode character escapes:

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 390)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

  \0 == nul
  \n == new line
  \r == carriage return
  \t == horizontal tab
  \u == small Unicode code points
  \U == large Unicode code points
  \x == raw ASCII byte (less than 0x80)
```

String Literals

Commentary

FIXME: Forcing + to concatenate strings is somewhat gross, a proper protocol would be better.

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 410)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

  string_literal   ::= ["]([^"\\\n\r]|character_escape|escape_expr)*["]
  escape_expr      ::= [\]escape_expr_body
  escape_expr_body ::= [(]escape_expr_body[)]
  escape_expr_body ::= [^\n\r"()]
```

`string_literal` tokens represent a string, and are surrounded by double quotes. String literals cannot span multiple lines.

String literals may contain embedded expressions in them (known as "interpolated expressions") subject to some specific lexical constraints: the expression may not contain a double quote ["], newline [n], or carriage return [r]. All parentheses must be balanced.

In addition to these lexical rules, an interpolated expression must satisfy the :ref:`expr <langref.expr>` production of the general swift grammar. This expression is evaluated, and passed to the constructor for the inferred type of the string literal. It is concatenated onto any fixed portions of the string literal with a global "+" operator that is found through normal name lookup.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 425); *backlink*

Unknown interpreted text role "ref".

```
// Simple string literal.
"Hello world!"

// Interpolated expressions.
"\(min)...\(max)" + "Result is \((4+i)*j)"
```

Identifier Tokens

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 445)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

  identifier ::= id-start id-continue*

  // An identifier can start with an ASCII letter or underscore...
  id-start ::= [A-Za-z_]

  // or a Unicode alphanumeric character in the Basic Multilingual Plane...
  // (excluding combining characters, which can't appear initially)
  id-start ::= [\u00A8\u00AA\u00AD\u00AF\u00B2-\u00B5\u00B7-00BA]
  id-start ::= [\u00BC-\u00BE\u00C0-\u00D6\u00D8-\u00F6\u00F8-\u00FF]
  id-start ::= [\u0100-\u02FF\u0370-\u167F\u1681-\u180D\u180F-\u1DBF]
  id-start ::= [\u1E00-\u1FFF]
  id-start ::= [\u200B-\u200D\u202A-\u202E\u203F-\u2040\u2054\u2060-\u206F]
  id-start ::= [\u2070-\u20CF\u2100-\u218F\u2460-\u24FF\u2776-\u2793]
  id-start ::= [\u2C00-\u2DFF\u2E80-\u2FFF]
  id-start ::= [\u3004-\u3007\u3021-\u302F\u3031-\u303F\u3040-\uD7FF]
  id-start ::= [\uF900-\uFD3D\uFD40-\uFDCF\uFDF0-\uFE1F\uFE30-FE44]
  id-start ::= [\uFE47-\uFFFD]

  // or a non-private-use, valid code point outside of the BMP.
  id-start ::= [\u10000-\u1FFFD\u20000-\u2FFFD\u30000-\u3FFFD\u40000-\u4FFFD]
  id-start ::= [\u50000-\u5FFFD\u60000-\u6FFFD\u70000-\u7FFFD\u80000-\u8FFFD]
  id-start ::= [\u90000-\u9FFFD\uA0000-\uAFFFD\uB0000-\uBFFFD\uC0000-\uCFFFD]
  id-start ::= [\uD0000-\uDFFFD\uE0000-\uEFFFD]

  // After the first code point, an identifier can contain ASCII digits...
  id-continue ::= [0-9]

  // and/or combining characters...
  id-continue ::= [\u0300-\u036F\u1DC0-\u1DFF\u20D0-\u20FF\uFE20-\uFE2F]

  // in addition to the starting character set.
  id-continue ::= id-start
```

```
identifier-or-any ::= identifier
identifier-or-any ::= '_'
```

The set of valid identifier characters is consistent with WG14 N1518, "Recommendations for extended identifier characters for C and C++". This roughly corresponds to the alphanumeric characters in the Basic Multilingual Plane and all non-private-use code points outside of the BMP. It excludes mathematical symbols, arrows, line and box drawing characters, and private-use and invalid code points. An identifier cannot begin with one of the ASCII digits '0' through '9' or with a combining character.

The Swift compiler does not normalize Unicode source code, and matches identifiers by code points only. Source code must be normalized to a consistent normalization form before being submitted to the compiler.

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 495)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

  // Valid identifiers
  foo
  _0
  swift
  vernissé
  闪亮
  מבורק
  

  // Invalid identifiers
  ☃      // Is a symbol
  0cool // Starts with an ASCII digit
  ◌́foo  // Starts with a combining character (U+0301)
  
      // Is a private-use character (U+F8FF)
```

Operator Tokens

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 517)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

  <a name="operator">operator</a> ::= [/=-+*%<>!&|^~]+
  <a name="operator">operator</a> ::= \.+

  <a href="#reserved_punctuation">Reserved for punctuation</a>: '.', '=', '->', and unary prefix '&'
  <a href="#whitespace">Reserved for comments</a>: '//', '/*' and '*/'

  operator-binary ::= operator
  operator-prefix ::= operator
  operator-postfix ::= operator

  left-binder  ::= [ \r\n\t\(\[\{,;:]
  right-binder ::= [ \r\n\t\)\]\},;:]

  <a name="any-identifier">any-identifier</a> ::= identifier | operator
```

`operator-binary`, `operator-prefix`, and `operator-postfix` are distinguished by immediate lexical context. An operator token is called *left-bound* if it is immediately preceded by a character matching `left-binder`. An operator token is called *right-bound* if it is immediately followed by a character matching `right-binder`. An operator token is an `operator-prefix` if it is right-bound but not left-bound, an `operator-postfix` if it is left-bound but not right-bound, and an `operator-binary` in either of the other two cases.

As an exception, an operator immediately followed by a dot ('.') is only considered right-bound if not already left-bound. This allows `a!.prop` to be parsed as `(a!).prop` rather than as `a ! .prop`.

The '`!`' operator is postfix if it is left-bound.

The '`?`' operator is postfix (and therefore not the ternary operator) if it is left-bound. The sugar form for `Optional` types must be left-bound.

When parsing certain grammatical constructs that involve '<' and '>' (such as protocol composition types), an `operator` with a leading '<' or '>' may be split into two or more tokens: the leading '<' or '>' and the remainder of the token, which may be an `operator` or `punctuation` token that may itself be further split. This rule allows us to parse nested constructs such as `A<B<C>>` without requiring spaces between the closing '>'s.

Implementation Identifier Token

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 565)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

  dollarident ::= '$' id-continue+
```

Tokens that start with a `$` are separate class of identifier, which are fixed purpose names that are defined by the implementation.

Escaped Identifiers

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 577)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

  identifier ::= '`' id-start id-continue* '`'
```

An identifier that would normally be a *keyword <langref.lexical.keyword>* may be used as an identifier by wrapping it in backticks '\`', for example:

```
func `class`() { /* ... */ }
let `type` = 0.type
```

Any identifier may be escaped, though only identifiers that would normally be parsed as keywords are required to be. The backtick-quoted string must still form a valid, non-operator identifier:

```
let `0` = 0       // Error, "0" doesn't start with an alphanumeric
let `foo-bar` = 0 // Error, '-' isn't an identifier character
let `+` = 0       // Error, '+' is an operator
```

Name Binding

Type Checking

Declarations

...

Import Declarations

...

`var` Declarations

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 624)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

  decl-var-head-kw ::= ('static' | 'class')? 'override'?
  decl-var-head-kw ::= 'override'? ('static' | 'class')?

  decl-var-head  ::= attribute-list decl-var-head-kw? 'var'

  decl-var       ::= decl-var-head pattern initializer?  (',' pattern initializer?)*

  // 'get' is implicit in this syntax.
  decl-var       ::= decl-var-head identifier ':' type brace-item-list

  decl-var       ::= decl-var-head identifier ':' type '{' get-set '}'

  decl-var       ::= decl-var-head identifier ':' type initializer? '{' willset-didset '}'

  // For use in protocols.
  decl-var       ::= decl-var-head identifier ':' type '{' get-set-kw '}'

  get-set        ::= get set?
  get-set        ::= set get

  get            ::= attribute-list ( 'mutating' | 'nonmutating' )? 'get' brace-item-list
  set            ::= attribute-list ( 'mutating' | 'nonmutating' )? 'set' set-name? brace-item-list
  set-name       ::= '(' identifier ')'

  willset-didset ::= willset didset?
  willset-didset ::= didset willset?

  willset        ::= attribute-list 'willSet' set-name? brace-item-list
  didset         ::= attribute-list 'didSet' set-name? brace-item-list

  get-kw         ::= attribute-list ( 'mutating' | 'nonmutating' )? 'get'
  set-kw         ::= attribute-list ( 'mutating' | 'nonmutating' )? 'set'
  get-set-kw     ::= get-kw set-kw?
  get-set-kw     ::= set-kw get-kw
```

`var` declarations form the backbone of value declarations in Swift. A `var` declaration takes a pattern and an optional initializer, and declares all the pattern-identifiers in the pattern as variables. If there is an initializer and the pattern is :ref:`fully-typed <langref.types.fully_typed>`, the initializer is converted to the type of the pattern. If there is an initializer and the pattern is not fully-typed, the type of initializer is computed independently of the pattern, and the type of the pattern is derived from the initializer. If no initializer is specified, the pattern must be fully-typed, and the values are default-initialized.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 661); *backlink*

Unknown interpreted text role "ref".

If there is more than one pattern in a `var` declaration, they are each considered independently, as if there were multiple declarations. The initial `attribute-list` is shared between all the declared variables.

A var declaration may contain a getter and (optionally) a setter, which will be used when reading or writing the variable, respectively.

Such a variable does not have any associated storage. A `var` declaration with a getter or setter must have a type (call it `T`). The getter function, whose body is provided as part of the `var-get` clause, has type `() -> T`. Similarly, the setter function, whose body is part of the `var-set` clause (if provided), has type `(T) -> ()`.

If the `var-set` or `willset` clause contains a `set-name` clause, the identifier of that clause is used as the name of the parameter to the setter or the observing accessor. Otherwise, the parameter name is `newValue`. Same applies to `didset` clause, but the default parameter name is `oldValue`.

FIXME: Should the type of a pattern which isn't fully typed affect the type-checking of the expression (i.e. should we compute a structured dependent type)?

Like all other declarations, `var`s can optionally have a list of :ref:`attributes <langref.decl.attribute_list>` applied to them.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 692); *backlink***
>
> Unknown interpreted text role "ref".

The type of a variable must be :ref:`materializable <langref.types.materializable>`. A variable is an lvalue unless it has a `var-get` clause but not `var-set` clause.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 695); *backlink***
>
> Unknown interpreted text role "ref".

Here are some examples of `var` declarations:

```
// Simple examples.
var a = 4
var b: Int
var c: Int = 42

// This decodes the tuple return value into independently named parts
// and both 'val' and 'err' are in scope after this line.
var (val, err) = foo()

// Variable getter/setter
var _x: Int = 0
var x_modify_count: Int = 0
var x1: Int {
  return _x
}
var x2: Int {
  get {
  return _x
  }
  set {
    x_modify_count = x_modify_count + 1
    _x = value
  }
}
```

Note that `get`, `set`, `willSet` and `didSet` are context-sensitive keywords.

`static` keyword is allowed inside structs and enums, and extensions of those.

`class` keyword is allowed inside classes, class extensions, and protocols.

Ambiguity 1

The production for implicit `get` makes this grammar ambiguous. For example:

```
class A {
  func get(_: () -> Int) {}
  var a: Int {
    get { return 0 } // Getter declaration or call to 'get' with a trailing closure?
  }
  // But if this was intended as a call to 'get' function, then we have a
  // getter without a 'return' statement, so the code is invalid anyway.
}
```

We disambiguate towards `get-set` or `willset-didset` production if the first token after `{` is the corresponding keyword, possibly preceded by attributes. Thus, the following code is rejected because we are expecting `{` after `set`:

```
class A {
  var set: Foo
  var a: Int {
    set.doFoo()
    return 0
  }
}
```

Ambiguity 2

The production with `initializer` and an accessor block is ambiguous. For example:

```
func takeClosure(_: () -> Int) {}
struct A {
  var willSet: Int
  var a: Int = takeClosure {
    willSet {} // A 'willSet' declaration or a call to 'takeClosure'?
  }
}
```

We disambiguate towards `willget-didset` production if the first token after `{` is the keyword `willSet` or

`didSet`, possibly preceded by attributes.

Rationale

Even though it is possible to do further checks and speculatively parse more, it introduces unjustified complexity to cover (hopefully rare) corner cases. In ambiguous cases users can always opt-out of the trailing closure syntax by using explicit parentheses in the function call.

`func` Declarations

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 798)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

  // Keywords can be specified in any order.
  decl-func-head-kw ::= ( 'static' | 'class' )? 'override'? ( 'mutating' | 'nonmutating' )?

  decl-func        ::= attribute-list decl-func-head-kw? 'func' any-identifier generic-params? func-signature br
```

`func` is a declaration for a function. The argument list and optional return value are specified by the type production of the function, and the body is either a brace expression or elided. Like all other declarations, functions are can have attributes.

If the type is not syntactically a function type (i.e., has no `->` in it at top-level), then the return value is implicitly inferred to be `()`. All of the argument and return value names are injected into the scope of the function body.</p>

A function in an extension of some type (or in other places that are semantically equivalent to an extension) implicitly get a `self` argument with these rules ... [todo]

`static` and `class` functions are only allowed in an extension of some type (or in other places that are semantically equivalent to an extension). They indicate that the function is actually defined on the metatype for the type, not on the type itself. Thus its implicit `self` argument is actually of metatype type.

`static` keyword is allowed inside structs and enums, and extensions of those.

`class` keyword is allowed inside classes, class extensions, and protocols.

TODO: Func should be an immutable name binding, it should implicitly add an attribute immutable when it exists.

TODO: Incoming arguments should be readonly, result should be implicitly writeonly when we have these attributes.

Function signatures

...

An argument name is a keyword argument if: - It is an argument to an initializer, or - It is an argument to a method after the first argument, or - It is preceded by a back-tick (`), or - Both a keyword argument name and an internal parameter name are specified.

`subscript` Declarations

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 854)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

  subscript-head ::= attribute-list 'override'? 'subscript' pattern-tuple '->' type

  decl-subscript ::= subscript-head '{' get-set '}'

  // 'get' is implicit in this syntax.
  decl-subscript ::= subscript-head brace-item-list

  // For use in protocols.
  decl-subscript ::= subscript-head '{' get-set-kw '}'
```

A subscript declaration provides support for subscripting an object of a particular type via a getter and (optional) setter. Therefore, subscript declarations can only appear within a type definition or extension.

The `pattern-tuple` of a subscript declaration provides the indices that will be used in the subscript expression, e.g., the `i` in `a[i]`. This pattern must be fully-typed. The `type` following the arrow provides the type of element being accessed, which must be materializable. Subscript declarations can be overloaded, so long as either the `pattern-tuple` or `type` differs from other declarations.

The `get-set` clause specifies the getter and setter used for subscripting. The getter is a function whose input is the type of the `pattern-tuple` and whose result is the element type. Similarly, the setter is a function whose result type is `()` and whose input is the type of the `pattern-tuple` with a parameter of the element type added to the end of the tuple; the name of the parameter is the `set-name`, if provided, or `value` otherwise.

```
// Simple bit vector with storage for 64 boolean values
struct BitVector64 {
  var bits: Int64

  // Allow subscripting with integer subscripts and a boolean result.
  subscript (bit : Int) -> Bool {
    // Getter tests the given bit
    get {
      return bits & (1 << bit)) != 0
    }

    // Setter sets the given bit to the provided value.
```

```
    set {
      var mask = 1 << bit
      if value {
        bits = bits | mask
      } else {
        bits = bits & ~mask
      }
    }
  }
}

var vec = BitVector64()
vec[2] = true
if vec[3] {
  print("third bit is set")
}
```

Attribute Lists

...

Types

...

Fully-Typed Types

...

Materializable Types

...

Patterns

> **Commentary**
>
> The pattern grammar mirrors the expression grammar, or to be more specific, the grammar of literals. This is because the conceptual algorithm for matching a value against a pattern is to try to find an assignment of values to variables which makes the pattern equal the value. So every expression form which can be used to build a value directly should generally have a corresponding pattern form.

> **System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 958)**
>
> Cannot analyze code. No Pygments lexer found for "none".
>
> ```
> .. code-block:: none
>
>   pattern-atom ::= pattern-var
>   pattern-atom ::= pattern-any
>   pattern-atom ::= pattern-tuple
>   pattern-atom ::= pattern-is
>   pattern-atom ::= pattern-enum-element
>   pattern-atom ::= expr
>
>   pattern      ::= pattern-atom
>   pattern      ::= pattern-typed
> ```

A pattern represents the structure of a composite value. Parts of a value can be extracted and bound to variables or compared against other values by *pattern matching*. Among other places, pattern matching occurs on the left-hand side of :ref:`var bindings <langref.decl.var>`, in the arguments of :ref:`func declarations <langref.decl.func>`, and in the <tt>case</tt> labels of :ref:`switch statements <langref.stmt.switch>`. Some examples:

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 970); *backlink***
>
> Unknown interpreted text role "ref".

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 970); *backlink***
>
> Unknown interpreted text role "ref".

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 970); *backlink***
>
> Unknown interpreted text role "ref".

```
var point = (1, 0, 0)

// Extract the elements of the "point" tuple and bind them to
// variables x, y, and z.
var (x, y, z) = point
print("x=\(x) y=\(y) z=\(z)")

// Dispatch on the elements of a tuple in a "switch" statement.
switch point {
case (0, 0, 0):
  print("origin")
// The pattern "_" matches any value.
case (_, 0, 0):
```

```
    print("on the x axis")
  case (0, _, 0):
    print("on the y axis")
  case (0, 0, _):
    print("on the z axis")
  case (var x, var y, var z):
    print("x=\(x) y=\(y) z=\(z)")
  }
```

A pattern may be "irrefutable", meaning informally that it matches all values of its type. Patterns in declarations, such as :ref:`var <langref.decl.var>` and :ref:`func <langref.decl.func>`, are required to be irrefutable. Patterns in the `case` labels of :ref:`switch statements <langref.stmt.switch>`, however, are not.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 1000); *backlink*

Unknown interpreted text role "ref".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 1000); *backlink*

Unknown interpreted text role "ref".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 1000); *backlink*

Unknown interpreted text role "ref".

The basic pattern grammar is a literal "atom" followed by an optional type annotation. Type annotations are useful for documentation, as well as for coercing a matched expression to a particular kind. They are also required when patterns are used in a :ref:`function signature <langref.decl.func.signature>`. Type annotations are currently not allowed in switch statements.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 1006); *backlink*

Unknown interpreted text role "ref".

A pattern has a type. A pattern may be "fully-typed", meaning informally that its type is fully determined by the type annotations it contains. Some patterns may also derive a type from their context, be it an enclosing pattern or the way it is used; this set of situations is not yet fully determined.

Typed Patterns

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 1023)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

  pattern-typed ::= pattern-atom ':' type
```

A type annotation constrains a pattern to have a specific type. An annotated pattern is fully-typed if its annotation type is fully-typed. It is irrefutable if and only if its subpattern is irrefutable.

Type annotations are currently not allowed in the `case` labels of `switch` statements; case patterns always get their type from the subject of the switch.

Any Patterns

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 1039)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

  pattern-any ::= '_'
```

The symbol _ in a pattern matches and ignores any value. It is irrefutable.

'var' and 'let' Patterns

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 1050)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

  pattern-var ::= 'let' pattern
  pattern-var ::= 'var' pattern
```

The `var` and `let` keywords within a pattern introduces variable bindings. Any identifiers within the subpattern bind new named variables to their matching values. 'var' bindings are mutable within the bound scope, and 'let' bindings are immutable.

```
var point = (0, 0, 0)
switch point {
// Bind x, y, z to the elements of point.
case (var x, var y, var z):
  print("x=\(x) y=\(y) z=\(z)")
}

switch point {
// Same. 'var' distributes to the identifiers in its subpattern.
case var (x, y, z):
  print("x=\(x) y=\(y) z=\(z)")
}
```

Outside of a <tt>var</tt> pattern, an identifier behaves as an :ref:`expression pattern <langref.pattern.expr>` referencing an existing definition.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 1075); *backlink*

Unknown interpreted text role "ref".

```
var zero = 0
switch point {
// x and z are bound as new variables.
// zero is a reference to the existing 'zero' variable.
case (var x, zero, var z):
  print("point off the y axis: x=\(x) z=\(z)")
default:
  print("on the y axis")
}
```

The left-hand pattern of a :ref:`var declaration <langref.decl.var>` and the argument pattern of a :ref:`func declaration <langref.decl.func>` are implicitly inside a `var` pattern; identifiers in their patterns always bind variables. Variable bindings are irrefutable.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 1090); *backlink*

Unknown interpreted text role "ref".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 1090); *backlink*

Unknown interpreted text role "ref".

The type of a bound variable must be :ref:`materializable <langref.types.materializable>` unless it appears in a :ref:`func-signature <langref.decl.func.signature>` and is directly of a `inout`-annotated type.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 1095); *backlink*

Unknown interpreted text role "ref".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 1095); *backlink*

Unknown interpreted text role "ref".

Tuple Patterns

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 1104)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

  pattern-tuple ::= '(' pattern-tuple-body? ')'
  pattern-tuple-body ::= pattern-tuple-element (',' pattern-tuple-body)* '...'?
  pattern-tuple-element ::= pattern
```

A tuple pattern is a list of zero or more patterns. Within a :ref:`function signature <langref.decl.func.signature>`, patterns may also be given a default-value expression.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 1110); *backlink*

Unknown interpreted text role "ref".

A tuple pattern is irrefutable if all its sub-patterns are irrefutable.

A tuple pattern is fully-typed if all its sub-patterns are fully-typed, in which case its type is the corresponding tuple type, where each `type-tuple-element` has the type, label, and default value of the corresponding `pattern-tuple-element`. A `pattern-tuple-element` has a label if it is a named pattern or a type annotation of a named pattern.

A tuple pattern whose body ends in `'...'` is a varargs tuple. The last element of such a tuple must be a typed pattern, and the type of that pattern is changed from `T` to `[T]`. The corresponding tuple type for a varargs tuple is a varargs tuple type.

As a special case, a tuple pattern with one element that has no label, has no default value, and is not varargs is treated as a grouping parenthesis: it has the type of its constituent pattern, not a tuple type.

`is` Patterns

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 1136)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

  pattern-is ::= 'is' type
```

`is` patterns perform a type check equivalent to the `x is T` cast operator. The pattern matches if the runtime type of a value is of the given type. `is` patterns are refutable and thus cannot appear in declarations.

```
class B {}
class D1 : B {}
class D2 : B {}

var bs : [B] = [B(), D1(), D2()]

for b in bs {
  switch b {
  case is B:
    print("B")
  case is D1:
    print("D1")
  case is D2:
    print("D2")
  }
}
```

Enum Element Patterns

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 1170)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

  pattern-enum-element ::= type-identifier? '.' identifier pattern-tuple?
```

Enum element patterns match a value of enum type if the value matches the referenced `case` of the enum. If the `case` has a type, the value of that type can be matched against an optional subpattern.

```
enum HTMLTag {
  case A(href: String)
  case IMG(src: String, alt: String)
  case BR
}

switch tag {
case .BR:
  print("<br>")
case .IMG(var src, var alt):
  print("<img src=\"\(escape(src))\" alt=\"\(escape(alt))\">")
case .A(var href):
  print("<a href=\"\(escape(href))\">")
}
```

Enum element patterns are refutable and thus cannot appear in declarations. (They are currently considered refutable even if the enum contains only a single `case`.)

Expressions in Patterns

Patterns may include arbitrary expressions as subpatterns. Expression patterns are refutable and thus cannot appear in declarations. An expression pattern is compared to its corresponding value using the `~=` operator. The match succeeds if `expr ~= value` evaluates to true. The standard library provides a default implementation of `~=` using `==` equality; additionally, range objects may be matched against integer and floating-point values. The `~=` operator may be overloaded like any function.

```
var point = (0, 0, 0)
switch point {
// Equality comparison.
case (0, 0, 0):
  print("origin")
// Range comparison.
case (-10...10, -10...10, -10...10):
  print("close to the origin")
default:
  print("too far away")
}

// Define pattern matching of an integer value to a string expression.
func ~=(pattern:String, value:Int) -&gt; Bool {
  return pattern == "\(value)"
}

// Now we can pattern-match strings to integers:
switch point {
case ("0", "0", "0"):
  print("origin")
default:
  print("not the origin")
}
```

The order of evaluation of expressions in patterns, including whether an expression is evaluated at all, is unspecified. The compiler is

free to reorder or elide expression evaluation in patterns to improve dispatch efficiency. Expressions in patterns therefore cannot be relied on for side effects.

Expressions

...

Function Application

...

Statements

...

`break` Statement

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 1271)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

  stmt-return ::= 'break'
```

The 'break' statement transfers control out of the enclosing 'for' loop or 'while' loop.

`continue` Statement

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 1283)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

    stmt-return ::= 'continue'
```

The 'continue' statement transfers control back to the start of the enclosing 'for' loop or 'while' loop.

...

`switch` Statement

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 1297)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

  stmt-switch ::= 'switch' expr-basic '{' stmt-switch-case* '}'

  stmt-switch-case ::= (case-label | default-label) brace-item+
  stmt-switch-case ::= (case-label | default-label) ';'

  case-label ::= 'case' pattern ('where' expr)? (',' pattern ('where' expr)?)* ':'
  default-label ::= 'default' ':'
```

'switch' statements branch on the value of an expression by :ref:`pattern matching <langref.pattern>`. The subject expression of the switch is evaluated and tested against the patterns in its `case` labels in source order. When a pattern is found that matches the value, control is transferred into the matching `case` block. `case` labels may declare multiple patterns separated by commas. Only a single `case` labels may precede a block of code. Case labels may optionally specify a *guard* expression, introduced by the `where` keyword; if present, control is transferred to the case only if the subject value both matches the corresponding pattern and the guard expression evaluates to true. Patterns are tested "as if" in source order; if multiple cases can match a value, control is transferred only to the first matching case. The actual execution order of pattern matching operations, and in particular the evaluation order of :ref:`expression patterns <langref.pattern.expr>`, is unspecified.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 1307); *backlink*

Unknown interpreted text role "ref".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 1307); *backlink*

Unknown interpreted text role "ref".

A switch may also contain a `default` block. If present, it receives control if no cases match the subject value. The `default` block must appear at the end of the switch and must be the only label for its block. `default` is equivalent to a final `case _` pattern. Switches are required to be exhaustive; either the contained case patterns must cover every possible value of the subject's type, or else an explicit `default` block must be specified to handle uncovered cases.

Every case and default block has its own scope. Declarations within a case or default block are only visible within that block. Case patterns may bind variables using the :ref:`var keyword <langref.pattern.var>`; those variables are also scoped into the corresponding case block, and may be referenced in the `where` guard for the case label. However, if a case block matches multiple patterns, none

of those patterns may contain variable bindings.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 1330); *backlink*

Unknown interpreted text role "ref".

Control does not implicitly 'fall through' from one case block to the next. :ref:`fallthrough statements <langref.stmt.fallthrough>` may explicitly transfer control among case blocks. :ref:`break <langref.stmt.break>` and :ref:`continue <langref.stmt.continue>` within a switch will break or continue out of an enclosing 'while' or 'for' loop, not out of the 'switch' itself.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 1337); *backlink*

Unknown interpreted text role "ref".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 1337); *backlink*

Unknown interpreted text role "ref".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 1337); *backlink*

Unknown interpreted text role "ref".

At least one `brace-item` is required in every case or default block. It is allowed to be a no-op. Semicolon can be used as a single no-op statement in otherwise empty cases in switch statements.

```
func classifyPoint(_ point: (Int, Int)) {
  switch point {
  case (0, 0):
    print("origin")

  case (_, 0):
    print("on the x axis")

  case (0, _):
    print("on the y axis")

  case (var x, var y) where x == y:
    print("on the y = x diagonal")

  case (var x, var y) where -x == y:
    print("on the y = -x diagonal")

  case (var x, var y):
    print("length \(sqrt(x*x + y*y))")
  }
}

switch x {
case 1, 2, 3:
  print("x is 1, 2 or 3")
default:
  ;
}
```

`fallthrough` Statement

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 1383)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

    stmt-fallthrough ::= 'fallthrough'
```

`fallthrough` transfers control from a `case` block of a :ref:`switch statement <langref.stmt.switch>` to the next `case` or `default` block within the switch. It may only appear inside a `switch`. `fallthrough` cannot be used in the final block of a `switch`. It also cannot transfer control into a `case` block whose pattern contains :ref:`var bindings <langref.pattern.var>`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 1387); *backlink*

Unknown interpreted text role "ref".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\archive\[swift-main][docs][archive]LangRefNew.rst, line 1387); *backlink*

Unknown interpreted text role "ref".

Standard Library

Commentary

It would be really great to have literate swift code someday, that way this could be generated directly from the code.

> This would also be powerful for Swift library developers to be able to depend on being available and standardized.

This describes some of the standard swift code as it is being built up. Since Swift is designed to give power to the library developers, much of what is normally considered the "language" is actually just implemented in the library.

All of this code is published by the '`swift`' module, which is implicitly imported into each source file, unless some sort of pragma in the code (attribute on an import?) is used to change or disable this behavior.

Builtin Module

In the initial Swift implementation, a module named `Builtin` is imported into every file. Its declarations can only be found by dot syntax. It provides access to a small number of primitive representation types and operations defined over them that map directly to LLVM IR.

The existence of and details of this module are a private implementation detail used by our implementation of the standard library. Swift code outside the standard library should not be aware of this library, and an independent implementation of the swift standard library should be allowed to be implemented without the builtin library if it desires.

For reference below, the description of the standard library uses the "`Builtin.`" namespace to refer to this module, but independent implementations could use another implementation if they so desire.

Simple Types

Void

```
// Void is just a type alias for the empty tuple.
typealias Void = ()
```

Int, Int8, Int16, Int32, Int64

> **Commentary**
>
> Having a single standardized integer type that can be used by default everywhere is important. One advantage Swift has is that by the time it is in widespread use, 64-bit architectures will be pervasive, and the LLVM optimizer should grow to be good at shrinking 64-bit integers to 32-bit in many cases for those 32-bit architectures that persist.

```
// Fixed size types are simple structs of the right size.
struct Int8  { value : Builtin.Int8 }
struct Int16 { value : Builtin.Int16 }
struct Int32 { value : Builtin.Int32 }
struct Int64 { value : Builtin.Int64 }
struct Int128 { value : Builtin.Int128 }

// Int is just an alias for the 64-bit integer type.
typealias Int = Int64
```

Int, Int8, Int16, Int32, Int64

```
struct Float  { value : Builtin.FPIEEE32 }
struct Double { value : Builtin.FPIEEE64 }
```

Bool, true, false

```
// Bool is a simple enum.
enum Bool {
  true, false
}

// Allow true and false to be used unqualified.
var true = Bool.true
var false = Bool.false
```

Arithmetic and Logical Operations

Arithmetic Operators

```
func * (lhs: Int, rhs: Int) -> Int
func / (lhs: Int, rhs: Int) -> Int
func % (lhs: Int, rhs: Int) -> Int
func + (lhs: Int, rhs: Int) -> Int
func - (lhs: Int, rhs: Int) -> Int
```

Relational and Equality Operators

```
func <  (lhs : Int, rhs : Int) -> Bool
func >  (lhs : Int, rhs : Int) -> Bool
func <= (lhs : Int, rhs : Int) -> Bool
func >= (lhs : Int, rhs : Int) -> Bool
func == (lhs : Int, rhs : Int) -> Bool
func != (lhs : Int, rhs : Int) -> Bool
```

Short Circuiting Logical Operators

```
func && (lhs: Bool, rhs: () -> Bool) -> Bool
func || (lhs: Bool, rhs: () -> Bool) -> Bool
```

Swift has a simplified precedence levels when compared with C. From highest to lowest:

```
"exponentiative:" <<, >>
"multiplicative:" *, /, %, &
"additive:" +, -, |, ^
"comparative:" ==, !=, <, <=, >=, >
"conjunctive:" &&
"disjunctive:" ||
```