Statements

TODO()

Id grammar-rule-statements not found

Id grammar-rule-statement not found

Unlike some other languages, Kotlin does not explicitly distinguish between statements, expressions and declarations, i.e., expressions and declarations can be used in statement positions. This section focuses only on those statements that are *not* expressions or declarations. For information on those parts of Kotlin, please refer to the [Expressions][Expressions] and [Declarations][Declarations] sections of the specification.

> Example: Kotlin supports using [conditionals][Conditional expression] both as expressions and as statements. As their use as expressions is more general, detailed information about conditionals is available in the [Expressions][Expressions] section of the specification.

Assignments

Id grammar-rule-assignment not found

Id grammar-rule-assignmentAndOperator not found

An *assignment* is a statement that writes a new value to some program entity, denoted by its left-hand side. Both left-hand and right-hand sides of an assignment must be expressions, more so, there are several restrictions for the expression on the left-hand side.

For an expression to be *assignable*, i.e. be allowed to occur on the left-hand side of an assignment, it **must** be one of the following:

- an identifier referring to a mutable property;
- a navigation expression referring to a mutable property;
- an indexing expression.

TODO(switch to navigation paths when we have them?)

> Note: Kotlin assignments **are not** expressions and cannot be used as such.

Simple assignments

A *simple assigment* is an assignment which uses the assign operator `=`. If the left-hand side of an assignment refers to a mutable property, a value of that

property is changed when an assignment is evaluated, using the following rules (applied in order).

- If a property is [delegated][Delegated property declaration], the corresponding operator function `setValue` is called using the right-hand side expression as the `value` argument;
- If a property has a [setter][Getters and setters], it is called using the right-hand side expression as its argument;
- Otherwise, if a property is a [mutable property][Mutable property declaration], its value is changed to the evaluation result of the right-hand side expression.

If the left-hand side of an assignment refers to a mutable property through the usage of safe navigation operator (`?.`), the same rules apply to it, but only if the left-hand side of the navigation operator is not referentially equal to `null` reference, e.g.:

```
a?.b?.z?.x = y
```

is semantically the same as

```
val __tmp = a?.b?.z
if(__tmp !== null) __tmp.x = y
```

TODO(just use setters for everything?)

If the left-hand side of an assignment is an indexing expression, the whole statement is treated as an [overloaded operator][Overloadable operators] with the following expansion:

$A[B_1, B_2, B_3, \ldots, B_N] = C$ is the same as calling A.set$(B_1, B_2, B_3, \ldots, B_N, C)$ where `set` is a suitable operator function.

Operator assignments

An *operator assignment* is a combined-form assignment which involves one of the following operators: `+=`, `-=`, `*=`, `/=`, `%=`. All of these operators are overloadable operator functions with the following expansions (applied in order):

- A`+=`B is exactly the same as one of the following:
  - A`.plusAssign(`B`)` if a suitable `plusAssign` operator function exists and is available;
  - A=A`.plus(`B`)` if a suitable `plus` operator function exists and is available.
- A`-=`B is exactly the same as one of the following:
  - A`.minusAssign(`B`)` if a suitable `minusAssign` operator function exists and is available;

  - A=A.`minus(`B`)` if a suitable `minus` operator function exists and is available.
- A`*=`B is exactly the same as one of the following:
  - A.`timesAssign(`B`)` if a suitable `timesAssign` operator function exists and is available;
  - A=A.`times(`B`)` if a suitable `times` operator function exists and is available.
- A`/=`B is exactly the same as one of the following:
  - A.`divAssign(`B`)` if a suitable `divAssign` operator function exists and is available;
  - A=A.`div(`B`)` if a suitable `div` operator function exists and is available;
- A`%=`B is exactly the same as one of the following:
  - A.`remAssign(`B`)` if a suitable `remAssign` operator function exists and is available;
  - A=A.`rem(`B`)` if a suitable `rem` operator function exists and is available.

  Note: as of Kotlin version 1.2.31, there are additional overloadable functions for `%` called `mod`/`modAssign`, which are deprecated.

After the expansion, the resulting [function call expression][Function call expressions] or simple assignment is processed according to their corresponding rules.

> Note: although for most real-world use cases operators `++` and `--` are similar to operator assignments, in Kotlin they are expressions and are described in the [corresponding section][Expressions] of this specification.

Loop statements

Loop statements describe an evaluation of a certain number of statements repeatedly until a *loop exit condition* applies.

Id grammar-rule-loopStatement not found

Loops are closely related to the semantics of [jump expressions][Jump expressions], as these expressions, namely [`break`][Break expression] and [`continue`][Continue expression], are only allowed in a body of a loop. Please refer to the corresponding sections for details.

While-loop statement

Id grammar-rule-whileStatement not found

A *while-loop statement* is similar to an [`if` expression][Conditional expression] in that it also has a condition expression and a body consisting of zero or more statements. While-loop statement evaluating its body repeatedly for as long as its condition expression evaluates to true or a [jump expression][Jump expressions] is evaluated to finish the loop.

> Note: this also means that the condition expression is evaluated before every evaluation of the body, including the first one.

The while-loop condition expression **must be a subtype** of `kotlin.Boolean`.

Do-while-loop statement

Id grammar-rule-doWhileStatement not found

A *do-while-loop statement*, similarly to a while-loop statement, also describes a loop, with the following differences. First, it has a different syntax. Second, it evaluates the loop condition expression **after** evaluating the loop body.

> Note: this also means that the body is always evaluated at least once.

The do-while-loop condition expression **must be a subtype** of `kotlin.Boolean`.

For-loop statement

Id grammar-rule-forStatement not found

> Note: unlike most other languages, Kotlin does not have a free-form condition-based for loops. The only form of a for-loop available in Kotlin is the "foreach" loop, which iterates over lists, arrays and other data structures.

A *for-loop statement* is a special kind of loop statement used to iterate over some data structure viewed as an iterable collection of elements. A for-loop statement consists of a loop body, a **container expression** and an **iteration variable declaration**.

The for-loop is actually an [overloadable][Overloadable operators] syntax form with the following expansion:

`for(VarDecl in C) Body` is the same as

```
val __iterator = C.iterator()
while (__iterator.hasNext()) {
    val VarDecl = __iterator.next()
    <... all the statements from Body>
}
```

where `iterator`, `hasNext`, `next` are all suitable operator functions available in the current scope. `VarDecl` here may be a variable name or a set of variable name as per [destructuring variable declarations][Destructuring declarations].

> Note: the expansion is hygienic, i.e., the generated iterator variable never clashes with any other variable in the program and cannot be accessed outside the expansion.

TODO(What about iterator value life-time and such?)

Code blocks

Id grammar-rule-block not found

Id grammar-rule-statements not found

A *code block* is a sequence of zero or more statements between curly braces separated by newlines or/and semicolons. Evaluating a code block means evaluating all its statements in the order they appear inside of it.

> Note: Kotlin does **not** support code blocks as statements; a curly-braces code block in a statement position is a [lambda literal][Lambda literals].

A *last expression* of a code block is the last statement in it (if any) if and only if this statement is also an expression. The last expressions are important when defining functions and control structure expressions.

A code block is said to contain no last expression if it does not contain any statements or its last statement is not an expression (e.g., it is an assignment, a loop or a declaration).

> Note: you may consider the case of a missing last expression as if a synthetic last expression with no runtime semantics and type `kotlin.Unit` is introduced in its place.

A *control structure body* is either a single statement or a code block. A *last expression* of a control structure body `CSB` is either the last expression of a code block (if `CSB` is a code block) or the single statement itself (if `CSB` is an expression). If a control structure body is not a code block or an expression, it has no last expression.

> Note: this is equivalent to wrapping the single statement in a new synthetic code block.

In some contexts, a control structure body is expected to have a value and/or a type. The value of a control structure body is:

- the value of its last expression if it exists;
- the singleton `kotlin.Unit` object otherwise.

The type of a control structure body is the type of its value.

TODO

- Labels
- Are declarations statements or not?
  - In the current grammar, they are
- How expansions with new variables actually work