Expressions

Glossary

CSB
: [Control structure body][Code blocks]

Introduction

TODO()

An expression may be *used as a statement* or *used as an expression* depending on the context. As all expressions are valid [statements][Statements], free expressions may be used as single statements or inside code blocks.

An expression is used as an expression, if it is encountered in any position where a statement is not allowed, for example, as an operand to an operator or as an immediate argument for a function call. An expression is used as a statement if it is encountered in any position where a statement is allowed.

Some expressions are only allowed to be used as statements, if certain restrictions are met; this may affect the semantics, the compile-time type information or/and the safety of these expressions.

TODO(strong/soft keywords?)

Constant literals

Constant literals are expressions which describe constant values. Every constant literal is defined to have a single standard library type, whichever it is defined to be on current platform. All constant literals are evaluated immediately.

Boolean literals

***BooleanLiteral*:**
: `true` | `false`

Keywords `true` and `false` denote boolean literals of the same values. These are strong keywords which cannot be used as identifiers unless [escaped][Escaped identifiers]. Values `true` and `false` always have the type `kotlin.Boolean`.

Integer literals

***IntegerLiteral*:**
DecDigitNoZero {*DecDigitOrSeparator*} *DecDigit* | *DecDigit*
***HexLiteral*:**
`0` (`x`|`X`) *HexDigit* {*HexDigitOrSeparator*} *HexDigit*
| `0` (`x`|`X`) *HexDigit*
***BinLiteral*:**
`0` (`b`|`B`) *BinDigit* {*BinDigitOrSeparator*} *BinDigit*
| `0` (`b`|`B`) *BinDigit*
***DecDigitNoZero*:**
DecDigit - `0`
***DecDigitOrSeparator*:**
DecDigit | *Underscore*
***HexDigitOrSeparator*:**
HexDigit | *Underscore*
***BinDigitOrSeparator*:**
BinDigit | *Underscore*
***DecDigits*:**
DecDigit {*DecDigitOrSeparator*} *DecDigit* | *DecDigit*

Decimal integer literals

A sequence of decimal digit symbols (`0` though `9`) is a decimal integer literal. Digits may be separated by an underscore symbol, but no underscore can be placed before the first digit or after the last one.

> Note: unlike other languages, Kotlin does not support octal literals. Even more so, any decimal literal starting with digit `0` and containing more than 1 digit is not a valid decimal literal.

Hexadecimal integer literals

A sequence of hexadecimal digit symbols (`0` through `9`, `a` through `f`, `A` through `F`) prefixed by `0x` or `0X` is a hexadecimal integer literal. Digits may be separated by an underscore symbol, but no underscore can be placed before the first digit or after the last one.

Binary integer literals

A sequence of binary digit symbols (`0` or `1`) prefixed by `0b` or `0B` is a binary integer literal. Digits may be separated by an underscore symbol, but no underscore can be placed before the first digit or after the last one.

The types for integer literals

Any of the decimal, hexadecimal or binary literals may be suffixed by the long literal mark (symbol `L`). An integer literal with the long literal mark has type `kotlin.Long`. A literal without the mark has a special [integer literal type][Integer literal types] dependent on the value of the literal:

- If the value is greater than maximum `kotlin.Long` value (see [built-in integer types][Built-in integer types]), it is an illegal integer literal and a compiler error;
- Otherwise, if the value is greater than maximum `kotlin.Int` value (see [built-in integer types][Built-in integer types]), it has type `kotlin.Long`;
- Otherwise, it has an integer literal type containing all the built-in integer types that are guaranteed to be able to represent this value.

  Note: for example, integer literal `0x01` has value 1 and therefore has type `LTS(kotlin.Byte, kotlin.Short, kotlin.Int, kotlin.Long)`. Integer literal `70000` has value 70000, which is not representable using types `kotlin.Byte` and `kotlin.Short` and therefore has type `LTS(kotlin.Int, kotlin.Long)`.

Real literals

***RealLiteral*:**
FloatLiteral | *DoubleLiteral*
***FloatLiteral*:**
DoubleLiteral (`f` | `F`) | *DecDigits* (`f` | `F`)
***DoubleLiteral*:**
[*DecDigits*] `.` *DecDigits* [*DoubleExponent*] | *DecDigits DoubleExponent*

A *real literal* consists of the following parts: the whole-number part, the decimal point (ASCII period character `.`), the fraction part and the exponent. Unlike other languages, Kotlin real literals may only be expressed in decimal numbers. A real literal may also be followed by a type suffix (`f` or `F`).

The exponent is an exponent mark (`e` or `E`) followed by an optionaly signed decimal integer (a sequence of decimal digits).

The whole-number part and the exponent part may be omitted. The fraction part may be omitted only together with the decimal point, if the whole-number part and either the exponent part or the type suffix are present. Unlike other languages, Kotlin does not support omitting the fraction part, but leaving the decimal point in.

The digits of the whole-number part or the fraction part or the exponent may be optionally separated by underscores, but an underscore may not be placed between, before, or after these parts. It also may not be placed before or after the exponent mark symbol.

A real literal without the type suffix has type `kotlin.Double`, a real literal with the type suffix has type `kotlin.Float`.

> Note: this means there is no special suffix associated with type `kotlin.Double`.

Character literals

***CharacterLiteral*:**
`'` (*EscapeSeq* | *<any character except CR, LF, ' and \>*) `'`
***EscapeSeq*:**
UnicodeCharacterLiteral | *EscapedCharacter*
***UnicodeCharacterLiteral*:**
`\ u` *HexDigit HexDigit HexDigit HexDigit*
***EscapedCharacter*:**
`\` (`t` | `b` | `r` | `n` | `'` | `"` | `\` | `$`)

A *character literal* defines a constant holding a unicode character value. A simply-formed character literal is any symbol between two single quotation marks (ASCII single quotation character `'`), excluding newline symbols (*CR* and *LF*), the single quotation mark itself and the escaping mark (ASCII backslash character `\`).

A character literal may also contain an escaped symbol of two kinds: a simple escaped symbol or a unicode codepoint. Simple escaped symbols include:

(TODO())
(TODO())
(TODO())
(TODO())
(TODO())
(TODO())

- `\t` — the unicode TAB symbol ;
- `\b` — the unicode BACKSPACE symbol ;
- `\r` — *CR*;
- `\n` — *LF*;
- `\'` — the unicode single quotation symbol ;
- `\"` — the unicode double quotation symbol ;
- `\\` — the unicode backslash symbol symbol ;
- `\$` — the unicode DOLLAR symbol .

A unicode codepoint escaped symbol is the symbol `\u` followed by exactly four hexadecimal digits. It represents the unicode symbol with the codepoint equal to the number represented by these four digits.

> Note: this means unicode codepoint escaped symbols support only unicode symbols in range from U+0000 to U+FFFF.

All character literals have type `kotlin.Char`.

String literals

Kotlin supports string interpolation which supersedes traditional string literals. For further details, please refer to the corresponding section.

Null literal

The keyword `null` denotes the **null reference**, which represents an absence of a value and is a valid value only for [nullable types][Nullable types]. Null reference has type [`kotlin.Nothing?`][`kotlin.Nothing`] and is, by definition, the only value of this type.

TODO(rearrange these sections)

Try-expression

Id grammar-rule-tryExpression not found

Id grammar-rule-catchBlock not found

Id grammar-rule-finallyBlock not found

A *try-expression* is an expression starting with the keyword `try`. It consists of a [code block][Code blocks] (*try body*) and one or more of the following kinds of blocks: zero or more *catch blocks* and an optional *finally block*. A *catch block* starts with the soft keyword `catch` with a single *exception parameter*, which is followed by a [code block][Code blocks]. A *finally block* starts with the soft keyword `finally`, which is followed by a [code block][Code blocks]. A valid try-expression must have at least one catch or finally block.

The try-expression evaluation evaluates its body; if any statement in the try body throws an exception (of type E), this exception, rather than being immediately propagated up the call stack, is checked for a matching catch block. If a catch block of this try-expression has an exception parameter of type $T :> E$, this catch block is evaluated immediately after the exception is thrown and the exception itself is passed inside the catch block as the corresponding parameter. If there are several catch blocks which match the exception type, the first one is picked.

TODO(Exception handling?)

If there is a finally block, it is evaluated after the evaluation of all previous try-expression blocks, meaning:

- If no exception is thrown during the evaluation of the try body, no catch blocks are executed, the finally block is evaluated after the try body, and the program execution continues as normal.
- If an exception was thrown, and one of the catch blocks matched its type, the finally block is evaluated after the evaluation of the matching catch block.

- If an exception was thrown, but no catch block matched its type, the finally block is evaluated before [propagating the exception][Exceptions] up the call stack.

The value of the try-expression is the same as the value of the [last expression][Code blocks] of the try body (if no exception was thrown) or the value of the last expression of the matching catch block (if an exception was thrown and matched). All other situations mean that an exception is going to be propagated up the call stack, and the value of the try-expression becomes irrelevant.

> Note: as desribed, the finally block (if present) is executed regardless, but it has no effect on the value returned by the try-expression.

The type of the try-expression is the [least upper bound][Least upper bound] of the types of the last expressions of the try body and the last expressions of all the catch blocks .

(TODO(not that simple))

> Note: these rules mean the try-expression always may be used as an expression, as it always has a corresponding result value.

Conditional expression

Id grammar-rule-ifExpression not found

Conditional expressions use a boolean value of one expression (*condition*) to decide which of the two [control structure bodies][Code blocks] (*branches*) should be evaluated. If the condition evaluates to `true`, the first branch (the *true branch*) is evaluated if it is present, otherwise the second branch (the *false branch*) is evaluated if it is present.

> Note: this means the following branchless conditional expression, despite being of almost no practical use, is valid in Kotlin
>
> ```
> if (condition) else;
> ```

The value of the resulting expression is the same as the value of the chosen branch.

The type of the resulting expression is the [least upper bound][Least upper bound] of the types of two branches , if both branches are present. If either of the branches are omitted, the resulting conditional expression has type [`kotlin.Unit`][`kotlin.Unit`] and may used only as a statement.

(TODO(not that simple))

TODO(Examples?)

The type of the condition expression must be a subtype of `kotlin.Boolean`, otherwise it is an error.

> Note: when used as expressions, conditional expressions are special w.r.t. operator precedence: they have the highest priority (the same

> as for all primary expressions) when placed on the right side of any binary expression, but when placed on the left side, they have the lowest priority. For details, see Kotlin [grammar][Syntax grammar].

When expression

Id grammar-rule-whenExpression not found

Id grammar-rule-whenEntry not found

Id grammar-rule-whenCondition not found

Id grammar-rule-rangeTest not found

Id grammar-rule-typeTest not found

When expression is similar to a conditional expression in that it allows one of several different [control structure bodies][Code blocks] (*cases*) to be evaluated, depending on some boolean conditions. The key difference is exactly that a when expressions may include several different conditions with their corresponding control structure bodies. When expression has two different forms: with bound value and without it.

When expression without bound value (the form where the expression enclosed in parentheses after the `when` keyword is absent) evaluates one of the different CSBs based on its condition from the *when entry*. Each when entry consists of a boolean *condition* (or a special `else` condition) and its corresponding CSB. When entries are checked and evaluated in their order of appearance. If the condition evaluates to `true`, the corresponding CSB is evaluated and the value of when expression is the same as the value of the CSB. All remaining conditions and expressions are not evaluated.

The `else` condition is a special condition which evaluates to `true` if none of the branches above it evaluated to `true`. The `else` condition **must** also be in the last when entry of when expression, otherwise it is a compile-time error.

> Note: informally, you can always replace the `else` condition with an always-`true` condition (e.g., boolean literal `true`) with no change to the resulting semantics.

When expression with bound value (the form where the expression enclosed in parentheses after the `when` keyword is present) are similar to the form without bound value, but use a different syntax for conditions. In fact, it supports three different condition forms:

- *Type test condition*: type checking operator followed by a type (`is T`). The resulting condition is a type check expression of the form `boundValue is T`.

- *Contains test condition*: containment operator followed by an expression (`in Expr`). The resulting condition is a containment check expression of the form `boundValue in Expr`.
- *Any other expression* (`Expr`). The resulting condition is an equality check of the form `boundValue == Expr`.
- The `else` condition, which is a special condition which evaluates to `true` if none of the branches above it evaluated to `true`. The `else` condition **must** also be in the last when entry of when expression, otherwise it is a compile-time error.

  Note: the rule for "any other expression" means that if a when expression with bound value contains a boolean condition, this condition is **checked for equality** with the bound value, instead of being used directly for when entry selection.

TODO(Examples)

(TODO(not that simple))

The type of the resulting expression is the [least upper bound][Least upper bound] of the types of all its entries . If the when expression is not exhaustive, it has type [`kotlin.Unit`][`kotlin.Unit`] and may used only as a statement.

Exhaustive when expressions

A when expression is called ***exhaustive*** if at least one of the following is true:

- It has an `else` entry;
- It has a bound value and at least one of the following is true:
  - The bound expression is of type `kotlin.Boolean` and the conditions contain both:
    - A [constant expression][Constant expressions] evaluating to `true`;
    - A [constant expression][Constant expressions] evaluating to `false`;
  - The bound expression is of a [`sealed class`][Sealed classes] type and all of its subtypes are covered using type test conditions in this expression. This should include checks for all direct subtypes of this sealed class. If any of the direct subtypes is also a sealed class, there should either be a check for this subtype or all its subtypes should be covered;
  - The bound expression is of an [`enum class`][Enum class declaration] type and all its enumerated values are checked for equality using constant expression;
  - The bound expression is of a [nullable type][Nullable types] T? and one of the cases above is met for its non-nullable counterpart T together with another condition which checks the bound value for equality with `null`.

TODO(Equality check with object behaves kinda like a type check. Or not.)

> Note: informally, an exhaustive when expression is guaranteed to evaluate one of its CSBs regardless of the specific when conditions.

Logical disjunction expression

Id grammar-rule-disjunction not found

Operator symbol `||` performs logical disjunction over two values of type `kotlin.Boolean`. This operator is **lazy**, meaning that it does not evaluate the right hand side argument unless the left hand side argument evaluated to `false`.

Both operands of a logical disjunction expression must have a type which is a subtype of `kotlin.Boolean`, otherwise it is a type error. The type of logical disjunction expression is `kotlin.Boolean`.

TODO(Types of errors? Compile-time, type, run-time, whatever?)

Logical conjunction expression

Id grammar-rule-conjunction not found

Operator symbol `&&` performs logical conjunction over two values of type `kotlin.Boolean`. This operator is **lazy**, meaning that it does not evaluate the right hand side argument unless the left hand side argument evaluated to `true`.

Both operands of a logical conjunction expression must have a type which is a subtype of `kotlin.Boolean`, otherwise it is a type error. The type of logical disjunction expression is `kotlin.Boolean`.

Equality expressions

Id grammar-rule-equality not found

Id grammar-rule-equalityOperator not found

Equality expressions are binary expressions involving equality operators. There are two kinds of equality operators: *reference equality operators* and *value equality operators*.

Reference equality expressions

Reference equality expressions are binary expressions which use reference equality operators: `===` and `!==`. These expressions check if two values are equal (`===`) or non-equal (`!==`) *by reference*: two values are equal by reference if and only if they represent the same runtime value.

For special values created without explicit constructor calls, notably, the constant literals and constant expressions composed of those literals, the following holds:

- If these values are non-equal by value, they are also non-equal by reference;
- Any instance of the null reference `null` is equal by reference to any other instance of the null reference;
- Otherwise, equality by reference is implementation-defined and must not be used as a means of comparing such values.

Reference equality expressions always have type `kotlin.Boolean`.

Value equality expressions

Value equality expressions are binary expressions which use value equality operators: `==` and `!=`. These operators are [overloadable][Overloadable operators] with the following expansion:

- `A == B` is exactly the same as `A?.equals(B) ?: (B === null)` where `equals` is a valid operator function available in the current scope;
- `A != B` is exactly the same as `!(A?.equals(B) ?: (B === null))` where `equals` is a valid operator function available in the current scope.

  Note: `kotlin.Any` type has a built-in open operator member function `equals`, meaning there is always at least one available overloading candidate for any value equality expression.

Value equality expressions always have type `kotlin.Boolean`. If the corresponding operator function `equals` has a different return type, it is a compile-time error.

Comparison expressions

Id grammar-rule-comparison not found

Id grammar-rule-comparisonOperator not found

Comparison expressions are binary expressions which use the comparison operators: `<`, `>`, `<=` and `>=`. These operators are [overloadable][Overloadable operators] with the following expansion:

- `A < B` is exactly the same as `A.compareTo(B) [<] 0`

- `A > B` is exactly the same as `0 [<] A.compareTo(B)`
- `A <= B` is exactly the same as `!(A.compareTo(B) [<] 0)`
- `A >= B` is exactly the same as `!(0 [<] A.compareTo(B))`

where `compareTo` is a valid operator function available in the current scope and `[<]` (read "boxed less") is a special operator unavailable in user-side Kotlin which performs integer "less-than" comparison of two integer numbers.

The `compareTo` operator function must have a return type `kotlin.Int`, otherwise it is a compile-time error.

All comparison expressions always have type `kotlin.Boolean`.

Type-checking and containment-checking expressions

Id grammar-rule-infixOperation not found

Id grammar-rule-inOperator not found

Id grammar-rule-isOperator not found

Type-checking expression

A type-checking expression uses a type-checking operator `is` or `!is` and has an expression E as a left-hand side operand and a type name T as a right-hand side operand. The type T must be [runtime-available][Runtime-available types], otherwise it is a compiler error. A type-checking expression checks whether the runtime type of E is a subtype of T for `is` operator, or not a subtype of T for `!is` operator.

Type-checking expression always has type `kotlin.Boolean`.

> Note: the expression `null is T?` for any type `T` always evaluates to `true`, as the type of the left-hand side (`null`) is `kotlin.Nothing?`, which is a subtype of any nullable type `T?`.
>
> Note: type-checking expressions may create [smart casts][Smart casts], for further details, refer to the corresponding section.

Containment-checking expression

A *containment-checking expression* is a binary expression which uses a containment operator `in` or `!in`. These operators are [overloadable][Overloadable operators] with the following expansion:

- `A in B` is exactly the same as `B.contains(A)`;
- `A !in B` is exactly the same as `!(B.contains(A))`.

where `contains` is a valid operator function available in the current scope.

> Note: this means that, contrary to the order of appearance in the code, the right-hand side expression of a containment-checking expression is evaluated before its left-hand side expression

The `contains` function must have a return type `kotlin.Boolean`, otherwise it is a compile-time error. Containment-checking expressions always have type `kotlin.Boolean`.

Elvis operator expression

Id grammar-rule-elvisExpression not found

An *elvis operator expression* is a binary expression which uses an elvis operator (`?:`). It checks whether the left-hand side expression is reference equal to `null`, and, if it is, evaluates and return the right-hand side expression.

This operator is **lazy**, meaning that if the left-hand side expression is not reference equal to `null`, the right-hand side expression is not evaluated.

The type of elvis operator expression is the [least upper bound][Least upper bound] of the non-nullable variant of the type of the left-hand side expression and the type of the right-hand side expression.

TODO(not that simple either)

Range expression

Id grammar-rule-rangeExpression not found

A *range expression* is a binary expression which uses a range operator `...` It is an [overloadable][Overloadable operators] operator with the following expansion:

- `A..B` is exactly the same as `A.rangeTo(B)`

where `rangeTo` is a valid operator function available in the current scope.

The return type of this function is not restricted. A range expression has the same type as the return type of the corresponding `rangeTo` overload variant.

Additive expression

Id grammar-rule-additiveExpression not found

Id grammar-rule-additiveOperator not found

An *additive expression* is a binary expression which uses the addition (`+`) or subtraction (`-`) operators. These are [overloadable][Overloadable operators] operators with the following expansions:

- `A + B` is exactly the same as `A.plus(B)`
- `A - B` is exactly the same as `A.minus(B)`

where `plus` or `minus` is a valid operator function available in the current scope.

The return type of these functions is not restricted. An additive expression has the same type as the return type of the corresponding operator function overload variant.

Multiplicative expression

Id grammar-rule-multiplicativeExpression not found

Id grammar-rule-multiplicativeOperator not found

A *multiplicative expression* is a binary expression which uses the multiplication (`*`), division (`/`) or remainder (`%`) operators. These are [overloadable][Overloadable operators] operators with the following expansions:

- `A * B` is exactly the same as `A.times(B)`
- `A / B` is exactly the same as `A.div(B)`
- `A % B` is exactly the same as `A.rem(B)`

where `times`, `div`, `rem` is a valid operator function available in the current scope.

> Note: as of Kotlin version 1.2.31, there exists an additional overloadable operator for `%` called `mod`, which is deprecated.

The return type of these functions is not restricted. A multiplicative expression has the same type as the return type of the corresponding operator function overload variant.

Cast expression

Id grammar-rule-asExpression not found

Id grammar-rule-asOperator not found

A *cast expression* is a binary expression which uses the cast operators `as` or `as?` and has the form `E as/as? T`, where E is an expression and T is a type name.

An **`as` cast expression** `E as T` is called *a unchecked cast* expression. This expression perform a runtime check whether the runtime type of E is a [subtype][Subtyping] of T and throws an exception otherwise. If type T is a [runtime-available][Runtime-available types] type without generic parameters, then this

exception is thrown immediately when evaluating the cast expression, otherwise it is platform-dependent whether an exception is thrown at this point.

TODO(We need to sort out undefined/implementation-defined/platform-defined)

> Note: even if the exception is not thrown when evaluating the cast expression, it is guaranteed to be thrown later when its result is used with any runtime-available type.

An unchecked cast expression result always has the same type as the type T specified in the expression.

An **`as?` cast expression** `E as? T` is called *a checked cast* expression. This expression is similar to the unchecked cast expression in that it also does a runtime type check, but does not throw an exception if the types do not match, it returns `null` instead. If type T is not a [runtime-available][Runtime-available types] type, then the check is not performed and `null` is never returned, leading to potential runtime errors later in the program execution. This situation should be reported as a compile-time warning.

> Note: if type T is a [runtime-available][Runtime-available types] type **with** generic parameters, type parameters are **not** checked w.r.t. subtyping. This is another porentially erroneous situation, which should be reported as a compile-time warning.

The checked cast expression type is the [nullable][Nullable types] variant of the type T.

> Note: cast expressions may create [smart casts][Smart casts], for further details, refer to the corresponding section.

Prefix expressions

Id grammar-rule-prefixUnaryExpression not found

Id grammar-rule-unaryPrefix not found

Id grammar-rule-prefixUnaryOperator not found

Annotated and labeled expression

Any expression in Kotlin may be prefixed with any number of [annotations][Annotations] and [labels][Labels]. These do not change the value of the expression and can be used by external tools and for implementing platform-dependent features.

Prefix increment expression

A *prefix increment* expression is an expression which uses the prefix form of operator `++`. It is an [overloadable][Overloadable operators] operator with the following expansion:

- `++A` is exactly the same as `A = A.inc(); A` where `inc` is a valid operator function available in the current scope.

  Note: informally, `++A` assigns the result of `A.inc()` to `A` and then returns `A` as the result.

For a prefix increment expression `++A` expression `A` must be [an assignable expression][Assignments]. Otherwise, it is a compile-time error.

A prefix increment expression has the same type as the return type of the corresponding `inc` overload variant.

> Note: as the result of `inc` is assigned to `A`, the return type of `inc` must be a subtype of `A`.

Prefix decrement expression

A *prefix decrement* expression is an expression which uses the prefix form of operator `--`. It is an [overloadable][Overloadable operators] operator with the following expansion:

- `--A` is exactly the same as `A = A.dec(); A` where `dec` is a valid operator function available in the current scope.

  Note: informally, `--A` assigns the result of `A.dec()` to `A` and then returns `A` as the result.

For a prefix decrement expression `--A` expression `A` must be [an assignable expression][Assignments]. Otherwise, it is a compile-time error.

A prefix decrement expression has the same type as the return type of the corresponding `dec` overload variant.

> Note: as the result of `dec` is assigned to `A`, the return type of `dec` must be a subtype of `A`.

Unary minus expression

An *unary minus* expression is an expression which uses the prefix form of operator `-`. It is an [overloadable][Overloadable operators] operator with the following expansion:

- `-A` is exactly the same as `A.unaryMinus()` where `unaryMinus` is a valid operator function available in the current scope.

No additional restrictions apply.

Unary plus expression

An *unary plus* expression is an expression which uses the prefix form of operator `+`. It is an [overloadable][Overloadable operators] operator with the following expansion:

- `+A` is exactly the same as `A.unaryPlus()` where `unaryPlus` is a valid operator function available in the current scope.

No additional restrictions apply.

Logical not expression

A *logical not* expression is an expression which uses the prefix operator `!`. It is an [overloadable][Overloadable operators] operator with the following expansion:

- `!A` is exactly the same as `A.not()` where `not` is a valid operator function available in the current scope.

No additional restrictions apply.

Postfix operator expressions

Id grammar-rule-postfixUnaryExpression not found

Id grammar-rule-postfixUnarySuffix not found

Id grammar-rule-postfixUnaryOperator not found

Postfix increment expression

A *postfix increment* expression is an expression which uses the postfix form of operator `++`. It is an [overloadable][Overloadable operators] operator with the following expansion:

- `A++` is exactly the same as `val $freshId = A; A = A.inc(); $freshId` where `inc` is a valid operator function available in the current scope.

  Note: informally, `A++` stores the value of A to a temporary variable, assigns the result of `A.inc()` to `A` and then returns the temporary variable as the result.

For a postfix increment expression `A++` expression `A` must be [assignable expressions][Assignable expressions]. Otherwise, it is a compile-time error.

A postfix increment expression has the same type as the return type of the corresponding `inc` overload variant.

> Note: as the result of `inc` is assigned to `A`, the return type of `inc` must be a subtype of `A`.

Postfix decrement expression

A *postfix decrement* expression is an expression which uses the postfix form of operator `--`. It is an [overloadable][Overloadable operators] operator with the following expansion:

- `A--` is exactly the same as `val $freshId = A; A = A.dec(); $freshId` where `dec` is a valid operator function available in the current scope.

  > Note: informally, `A--` stores the value of A to a temporary variable, assigns the result of `A.dec()` to `A` and then returns the temporary variable as the result.

For a postfix decrement expression `A--` expression `A` must be [assignable expressions][Assignable expressions]. Otherwise, it is a compile-time error.

A postfix decrement expression has the same type as the return type of the corresponding `dec` overload variant.

> Note: as the result of `dec` is assigned to `A`, the return type of `dec` must be a subtype of `A`.

Not-null assertion expression

TODO(We need to define what “evaluation” is)

A *not-null assertion expression* is a postfix expression which uses an operator `!!`. For an expression `e!!`, if the type of `e` is nullable, a not-null assertion expression checks, whether the evaluation result of `e` is equal to `null` and, if it is, throws a runtime exception. If the evaluation result of `e` is not equal to `null`, the result of `e!!` is the evaluation result of `e`.

If the type of `e` is non-nullable, not-null assertion expression `e!!` has no effect.

The type of non-null assertion expression is the [non-nullable][Nullable types] variant of the type of `e`.

> Note: this type may be non-denotable in Kotlin and, as such, may be [approximated][Type approximation] in some situations with the help of [type inference][Type inference].

TODO(Example)

Indexing expressions

Id grammar-rule-postfixUnaryExpression not found

Id grammar-rule-postfixUnarySuffix not found

Id grammar-rule-indexingSuffix not found

An *indexing expression* is a suffix expression which uses one or more subexpression as *indices* between square brackets (`[` and `]`).

It is an [overloadable][Overloadable operators] operator with the following expansion:

- `A[I_0,I_1,...,I_N]` is exactly the same as `A.get(I_0,I_1,...,I_N)`, where `get` is a valid operator function available in the current scope.

An indexing expression has the same type as the corresponding `get` expression.

Indexing expressions are [assignable][Assignable expressions]. For a corresponding assignment form, see [indexing assignment][Indexing assignment].

Call and property access expressions

Id grammar-rule-postfixUnaryExpression not found

Id grammar-rule-postfixUnarySuffix not found

Id grammar-rule-navigationSuffix not found

Id grammar-rule-callSuffix not found

Id grammar-rule-annotatedLambda not found

Id grammar-rule-valueArguments not found

Id grammar-rule-typeArguments not found

Id grammar-rule-typeProjection not found

Id grammar-rule-typeProjectionModifiers not found

Id grammar-rule-memberAccessOperator not found

Navigation operators

Expressions which use the navigation binary operators (`.`, `.?` or `::`) are syntactically similar, but, in fact, may have very different semantics.

`a.c` may have one of the following semantics when used as an expression:

- A fully-qualified type, property or object name. The left side of `.` must be a package name, while the right side corresponds to a declaration in that package.

  Note: qualification uses operator `.` only.

- A property access. Here `a` is a value available in the current scope and `c` is a property name.
- A function call if followed by the call suffix (arguments in parentheses). Here `a` is a value available in the current scope and `c` is a function name. These expressions follow the [overloading][Overload resolution] rules.

`a::c` may have one of the following semantics when used as an expression:

- A class literal expression if, instead of an identifier, `c` is the keyword `class`;
- A property reference. Here `a` may be either a value available in the current scope or a type name, and `c` is a property name.
- A function reference. Here `a` may be either a value available in the current scope or a type name, and `c` is a function name.

`a?.c` is a *safe navigation* operator, which has the following expansion:

- `a?.c` is exactly the same as `if (a != null) a.c else null`.

  Note: this means the type of `a?.c` is the [nullable][Nullable types] variant of the type of `a.c`.

Callable references

TODO(this is a stub)

Callable references are a special kind of expressions used to refer to callables (properties and functions) without actually calling/accessing them. It is not to be confused with class literals that use similar syntax, but with the keyword `class` used instead of the identifier.

A callable reference `A::c` where `A` is a type name and `c` is a name of a callable available for type `A` is a *callable reference* for a type. A callable reference `e::c` where `e` is another expression and `c` is a name of a callable available for type `A` is a *callable reference* for expression `e`. The exact callable selected when using this syntax is based on [overload resolution][Overload resolution] much like when accessing the value of a property using the usual navigation syntax.

Depending on the meaning of the left-hand and right-hand sides of the expressions, the value of the expression is different:

- If the left-hand side of the expression is a type, but is not a value (an example of a type which is also used as a value is an object type), while the right-hand side of the expression is resolved to refer to a property of the type on the left-hand side, then the expression is a type-property reference;
- If the left-hand side of the expression is a type, but is not a value (an example of a type which is also used as a value is an object type), while the right-hand side of the expression is resolved to refer to a function available for a receiver of the type on the left-hand side, then the expression is a type-function reference;
- If the left-hand side of the expression is a value, while the right-hand side of the expression is resolved to refer to a property of the value on the left-hand side, then the expression is a value-property reference;
- If the left-hand side of the expression is a value, while the right-hand side of the expression is resolved to refer to a function for the receiver being th value on the left-hand side, then the expression is a value-function reference.

The types of these expressions are implementation-defined, but the following constraints must hold:

- The type of any kind of property reference is a subtype of `kotlin.reflect.KProperty<T>`, where the type parameter `T` is fixed to the type of the property;
- The type of any kind of callable reference is a subtype of [function type][Function types] that allows the corresponding callable to be accessed/called accordingly:
  - For a type-callable reference, it is an extension function type `O.(Arg0 ... ArgN) -> R`, where `O` is a receiver type same as the left-hand type of the expression, `Arg0, ... , ArgN` are either empty (for a property reference) or are the types of function formal parameters (for a function reference) and `R` is the result type of the callable;
  - For a value-callable reference, it is a normal function type `(Args) -> R`, where `Arg0, ... , ArgN` are either empty (for a property reference) or are the types of function formal parameters (for a function reference) and `R` is the result type of the callable. The receiver is bound to the left-hand side expression of the reference expression.

Being of an appropriate function type also means that the values defined by these references are valid callables themselves, with an appropriate `operator invoke` overload, that allows using call syntax to evaluate the value of the callable with the appropriate arguments.

> Note: one may say that any function reference is essentially the same as a lambda literal with the corresponding number of arguments, calling the callable being referenced.

TODO(this is pretty complex, actually. Do we need all the K(Mutable)PropertyN business defined in the specification???) TODO(we need to update overload resolution section with these guys)

Class literals

A class literal is very similar in syntax to a callable reference, with the difference being that it uses the keyword `class` instead of the referenced identifier. Similar to callable references, there are two forms of class literals: with a type used as the left-hand side argument of the expression and with another expression used as such. This is also one of the few cases where a parameterized type may (**and must**) be used without its type parameters.

All class literals have type `kotlin.KClass<T>` and produce a platform-defined object associated with type `T`, which, in turn, is either the type given as the left-hand side of the expression or the [runtime type][Runtime type information] of the value given as the left-hand side of the expression. In both cases, `T` must be a [runtime-available type][Runtime type information] in the current scope. As the runtime type of the expression is not known at compile time, the compile-time type of the expression is `kotlin.KClass<U>` where $T <: U$ and `U` is the compile-time of the expression.

The produced object can be used to allow access to platform-specific capabilities of the runtime type information available on particular platform, either directly or through reflection facilities.

TODO(this is a stub)

TODO(Identifiers, paths, that kinda stuff)

Function literals

Kotlin supports using functions as values. This includes, among other things, being able to use named functions (via function references) as parts of expressions. Sometimes it does not make much sense to provide a separate function declaration, but rather define a function in-place. This is implemented using *function literals*.

There are two types of function literals in Kotlin: *lambda literals* and *anonymous function declarations*. Both of these provide a way of defining a function in-place, but have subtle differences.

> Note: as some may consider function literals to be closely related to function declarations, [here][Function declaration] is the corresponding section of the specification.

Anonymous function declarations

Id grammar-rule-anonymousFunction not found

Anonymous function declarations, despite their name, are not declarations per se, but rather expressions which resemble function declarations. They have a syntax very similar to function declarations, with the following key differences:

- Anonymous functions do not have a name;
- Anonymous functions may not have type parameters;
- Anonymous functions may not have default parameters;
- Anonymous functions may have variable argument parameters, but they are automatically decayed to non-variable argument parameters of the corresponding array type .

(TODO(how does this really work?))

Anonymous function declaration may declare an anonymous extension function.

> Note: as anonymous functions may not have type parameters, you cannot declare an anonymous extension function on a parameterized receiver type.

The type of an anonymous function declaration is the function type constructed similarly to a [named function declaration][Function declaration].

Lambda literals

Id grammar-rule-lambdaLiteral not found

Id grammar-rule-lambdaParameters not found

Id grammar-rule-lambdaParameter not found

Lambda literals are similar to anonymous function declarations in that they define a function with no name. Lambda also use very different syntax, similar to control structure bodies of other expressions.

Every lambda literal consists of an optional lambda parameter list, specified before the arrow (`->`) operator and a body, which is everything after the arrow operator. Lambda body introduces a new statement scope.

Lambda literals has the same restrictions as anonymous function declarations, but also cannot have `vararg` parameters. They can, however, introduce destructuring parameters similar to destructuring property declarations.

TODO(destructuring lambda parameters)

If a lambda expression has no parameter list, it can actually be defining an anonymous function with either zero or one parameter, the exact case dependent on the context of the usage of this expression. The selection of number of parameters in this case is performed during [type inference][Type inference]. Any

lambda may also define either a normal function or an expansion function, the exact case also dependent on the context of the usage of lambda expression.

If the lambda expression has no parameter list, but has one parameter, this parameter can be accessed inside the lambda body using a special property called `it`. If the lambda expression defines an expansion function, the expansion receiver may be accessed using standard `this` syntax inside the lambda body.

> Note: having no parameter list (and no arrow operator) in a lambda is different from having zero parameters (nothing preceding the arrow operator).

Lambda literals are different from other forms of function definition in that the `return` expressions inside lambda body, unless qualified, refers to the outside non-lambda function the expression is used in rather than the lambda expression itself. Such returns are only allowed if the function defined by the lambda is guaranteed to be [inlined][Inlining] and are not allowed at all otherwise.

If the lambda expression is labeled, it can also be returned from using the [labeled return expression][Labeled return expression]. In addition to this, if the lambda expression is used as a trailing lambda parameter to a function call, the name of the function used in the call may be used instead of the label. If a particular labeled `return` expression is used inside multiple lambda bodies invoked during the call of the same function, this is an ambiguity and should be reported as a compile-time error.

TODO(Typing)

Any properties used in any way inside the lambda body are **captured** by the lambda expression and, depending on whether it is inlined or not, affect how this properties are processed by other mechanisms, e.g. [smart casts][Smart casts].

TODO(Rules of capturing)

Object literals

Id grammar-rule-objectLiteral not found

Object literals are used to define anonymous objects in Kotlin. Anonymous objects are similar to regular objects, but they (obviously) have no name and thus can be used only as expressions. Anonymous objects, just like regular object declarations, can have at most one base class and zero or more base interfaces declared in its supertype specifiers.

The main difference between the regular object declaration and an anonymous object is its type. The type of an anonymous object is a special kind of type which is usable (and visible) only in the scope where it is declared. It is similar

to a type of a regular object declaration, but, as it cannot be used outside the scope, with some interesting effects.

When a value of an anonymous object type escapes current scope:

- If the type has only one declared supertype, it is implicitly downcasted to this declared supertype;
- If the type has several declared supertypes, there must be an implicit or explicit cast to any suitable type visible outside the scope, otherwise it is a compile-time error.

  Note: an implicit cast may arise, for example, from the results of the type inference.

  Note: in this context "escaping" current scope is performed immediately if the corresponding value is declared as a global- or classifier-scope property, as those are a part of package interface.

TODO: This is more complex. From D.Petrov's comment:

> This is a bit more complex for anonymous object return types of private functions and properties:
>
> ```
> class M {
> private fun foo() = object {
> fun bar() { println("foo.bar") }
> }
>
> fun test1() = foo().bar()
> fun test2() = foo()
> }
>
> fun main() {
> M().test1() // OK, prints "foo.bar"
> M().test2().bar() // Error: Unresolved reference: bar
> }
> ```

This-expressions

Id grammar-rule-thisExpression not found

This-expressions are special kind of expressions used to access [receivers][Receivers] available in current scope. The basic form of this expression, denoted by `this` keyword, is used to access the current implicit receiver according to the receiver overloading rules. In order to access other receivers, labeled `this` expressions are used. These may be any of the following:

- `this@type`, where `type` is a name of any classifier currently being declared (that is, this-expression is located in the inner scope of the classifier declaration), refers to the implicit object of the type being declared;
- `this@function`, where `function` is a name of any extension function currently being declared (that is, this-expression is located in the function body), refers to the implicit receiver object of the extension function;
- `this@lambda`, where `lambda` is a [label][Labels] provided for a lambda literal currently being declared (that is, this-expression is located in the lambda expression body), refers to the implicit receiver object of the lambda expression.

Any other form of this-expression is illegal and must be a compile-time error.

Super-forms

Id grammar-rule-superExpression not found

Super-forms are special kind of expression which can only be used as receivers in a function or property access expression. Any use of super-form expression in any other context is a compile-time error.

Super-forms are used in classifier declarations to access method implementations from the supertypes without invoking overriding behaviour.

TODO(The rest...)

Jump expressions

Id grammar-rule-jumpExpression not found

Jump expressions are expressions which redirect the evaluation of the program to a different program point. All these expressions have several things in common:

- They all have type [`kotlin.Nothing`][`kotlin.Nothing`], meaning that they never produce any runtime value;
- Any code which follows such expressions is never evaluated.

Throw expressions

TODO([Exceptions] go first)

Return expressions

A *return expression*, when used inside a function body, immediately stops evaluating the current function and returns to its caller, effectively making the function call expression evaluate to the value specified in this return expression (if any). A return expression with no value implicitly returns the `kotlin.Unit` object.

There are two forms of return expression: a simple return expression, specified using the `return` keyword, which returns from the innermost [function declaration][Function declaration] (or Anonymous function declaration) and a labeled return expression of the form `return@Context` where `Context` may be one of the following:

- The name of one of the enclosing function declarations, which refers to this function. If several declarations match one name, it is a compile-time error;
- If `return@Context` is inside a lambda expression body, the name of the function **using** this lambda expression as its argument may be used as `Context` to refer to the lambda literal itself.
- TODO(return from a labeled lambda)

  Note: these rules mean that a simple return expression inside a lambda expression returns **from the innermost function**, in which this lambda expression is defined.

If returning from the referred function is allowed in the current context, the return is performed as usual. If returning from the referred function is not allowed, it is a compile-time error.

TODO(What does it mean for returns to be disallowed?)

Continue expression

A *continue expression* is a jump expression allowed only within loop bodies. When evaluated, this expression passes the control to the start of the next loop iteration (aka "continue-jumps").

There are two forms of continue expressions:

- A simple continue expression, specified using the `continue` keyword, which continue-jumps to the innermost loop statement in the current scope;
- A labeled continue expression, denoted `continue@Loop`, where `Loop` is a label of a labeled loop statement `L`, which continue-jumps to the loop `L`.

Future use: as of Kotlin 1.2.60, a simple continue expression is not allowed inside `when` expressions.

Break expression

A *break expression* is a jump expression allowed only within loop bodies. When evaluated, this expression passes the control to the next program point immediately after the loop (aka "break-jumps").

There are two forms of break expressions:

- A simple break expression, specified using the `break` keyword, which break-jumps to the innermost loop statement in the current scope;
- A labeled break expression, denoted `break@Loop`, where `Loop` is a label of a labeled loop statement `L`, which break-jumps to the loop `L`.

  Future use: as of Kotlin 1.2.60, a simple break expression is not allowed inside when expressions.

String interpolation expressions

Id grammar-rule-stringLiteral not found

Id grammar-rule-lineStringLiteral not found

Id grammar-rule-multiLineStringLiteral not found

Id grammar-rule-lineStringContent not found

Id grammar-rule-lineStringExpression not found

Id grammar-rule-multiLineStringContent not found

Id grammar-rule-multiLineStringExpression not found

String interpolation expressions replace the traditional string literals and supersede them. A string interpolation expression consists of one or more fragments of two different kinds: string content fragments (raw pieces of string content found inside the quoted literal) and *interpolated expressions*, delimited by the special syntax using the `$` symbol. This syntax allows to specify such fragments by directly following the `$` symbol with either a single identifier (if the expression is a single identifier) or a control structure body. In either case, the interpolated value is evaluated and converted into a `kotlin.String` by a process defined below. The resulting value of a string interpolation expression is the joining of all fragments in the expression.

An interpolated value v is converted to `kotlin.String` according to the following convention:

- If it is equal to the null reference, the result is `"null"`;

- Otherwise, the result is v`.toString()` where `toString` is the `kotlin.Any` member function (no overloading resolution is performed to choose the function in this context).

There are two kinds of string interpolation expressions: line interpolation expressions and multiline (or raw) interpolation expressions. The difference is that some symbols (namely, newline symbols) are not allowed to be used inside line interpolation expressions and they need to be escaped (see [grammar][Grammar] for details). On the other hand, multiline interpolation expressions allow such symbols inside them, but do not allow single character escaping of any kind.

> Note: among other things, this means that the escaping of the `$` symbol is impossible in multiline strings. If you need an escaped `$` symbol, use an interpolation fragment instead: `"${'$'}"`

String interpolation expression always has type `kotlin.String`.

TODO(define this using actual `kotlin.StringBuilder` business?)

TODO(list all the allowed escapes here?)

TODOs()

- Class literals
- Smart casts vs compile-time types
- What does `decaying` for vararg actually mean?
- Where to define spread operator?
- Object literal types look just like restricted union types. Are there any traps hidden here?
- The last paragraph in object literals is also pretty shady