Kotlin language specification

Version 1.9-rfc+0.1

Marat Akhin Mikhail Belyaev

Chapter 8

Expressions

Glossary

CSB
Control structure body

Introduction

Expressions (together with statements) are one of the main building blocks of any program, as they represent ways to *compute* program values or *control* the program execution flow.

In Kotlin, an expression may be *used as a statement* or *used as an expression* depending on the context. As all expressions are valid statements, standalone 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 allowed to be used as statements, only if certain restrictions are met; this may affect the semantics, the compile-time type information or/and the safety of these expressions.

8.1 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.

8.1.1 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. Values `true` and `false` always have the type `kotlin.Boolean`.

8.1.2 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*:**
  `'1'` | `'2'` | `'3'` | `'4'` | `'5'` | `'6'` | `'7'` | `'8'` | `'9'`

***DecDigitOrSeparator*:**
  DecDigit | `'_'`

***HexDigitOrSeparator*:**
  HexDigit | `'_'`

BinDigitOrSeparator
  BinDigit | `'_'`

***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.

8.1.3 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 dependent on the value of the literal:

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

  Example: integer literal `0x01` has value 1 and therefore has type ILT(`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 ILT(`kotlin.Int`, `kotlin.Long`).

8.1.4 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 optionally 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`.

8.1.5 Character literals

CharacterLiteral
  `'''` (*EscapeSeq* | <any character excluding CR, LF, `'''` or `'\'`>) `'''`

EscapeSeq
  UniCharacterLiteral | *EscapedIdentifier*

UniCharacterLiteral
  `'\'` `'u'` *HexDigit HexDigit HexDigit HexDigit*

EscapedIdentifier
  `'\'` (`'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 `\`).

All character literals have type `kotlin.Char`.

Escaped characters

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

- `\t` — the Unicode TAB symbol (U+0009);

- `\b` — the Unicode BACKSPACE symbol (U+0008);
- `\r` — *CR*;
- `\n` — *LF*;
- `\'` — the Unicode apostrophe symbol (U+0027);
- `\"` — the Unicode double quotation symbol (U+0028);
- `\\` — the Unicode backslash symbol (U+005C);
- `\$` — the Unicode DOLLAR sign (U+0024).

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.

8.1.6 String literals

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

8.1.7 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. Null reference has type `kotlin.Nothing?` and is, by definition, the only value of this type.

8.2 Constant expressions

We use the term "constant expression" to refer to any expression constructed of the following:

- constant literals
- access expressions to enum entries
- string interpolation over constant expressions
- an implementation-defined set of functions that can always be evaluated at compile-time

8.3 String interpolation expressions

***stringLiteral*:**
  lineStringLiteral
  | *multiLineStringLiteral*

***lineStringLiteral*:**
  `'"'` {*lineStringContent* | *lineStringExpression*} `'"'`

***multiLineStringLiteral*:**
`'"""'` {*multiLineStringContent* | *multiLineStringExpression* | `'"'`}
TRIPLE_QUOTE_CLOSE

***lineStringContent*:**
LineStrText
| *LineStrEscapedChar*
| *LineStrRef*

***lineStringExpression*:**
`'${'`
{*NL*}
expression
{*NL*}
`'}'`

***multiLineStringContent*:**
MultiLineStrText
| `'"'`
| *MultiLineStrRef*

***multiLineStringExpression*:**
`'${'`
{*NL*}
expression
{*NL*}
`'}'`

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 inside the quoted literal) and *interpolated expression fragments*, specified by a special syntax using the `$` symbol.

Interpolated expressions support two different forms.

- `$id`, where `id` is a simple path available in the current scope;
- `${e}`, where `e` is a valid Kotlin expression.

  Note: the first form requires `id` to be a simple path; if you want to reference a qualified path (e.g., `foo.bar`), you should use the second form as `${foo.bar}`.

In either case, the interpolated value is evaluated and converted into `kotlin.String` by a process defined below. The resulting value of a string interpolation expression is the concatenation 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 this 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 in the same way they are escaped in character literals. 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 escaping of the `$` symbol is impossible in multiline strings. If you need an escaped `$` symbol, use an interpolated expression `"${'$'}"` instead.

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

> Examples:
>
> The following code
>
> ```
> val a = "Hello, $x is ${foo()}"
> val b = """
> Hello, $x
> is "${foo()}"
> """
> ```
>
> is equivalent to
>
> ```
> val a = "Hello, " + (x?.toString() ?: "null") +
>         " is " + (foo()?.toString() ?: "null")
> val b =  "\nHello, " + (x?.toString() ?: "null") +
>          "\nis \"" + (foo()?.toString() ?: "null") + "\"\n"
> ```

8.4 Try-expressions

***tryExpression*:**
 `'try'` {*NL*} *block* ((({*NL*} *catchBlock* {{*NL*} *catchBlock*}) [{*NL*} *finallyBlock*]) | ({*NL*} *finallyBlock*))

***catchBlock*:**
 `'catch'`
 {*NL*}
 `'('`
 {*annotation*}
 simpleIdentifier
 `':'`
 type

    [{*NL*} `','`]
    `')'`
    {*NL*}
    block

***finallyBlock*:**
    `'finally'` {*NL*} *block*

A *try-expression* is an expression starting with the keyword `try`. It consists of a code block (*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. A *finally block* starts with the soft keyword `finally`, which is followed by a code block. 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.

For an in-detail explanation on how exceptions and catch-blocks work, please refer to the Exceptions section. For a low-level explanation, please refer to the platform-specific parts of this document.

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 up the call stack.

The value of the try-expression is the same as the value of the last expression 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 is undefined.

> Note: as described, the finally block (if present) is always executed, but has no effect on the value of the try-expression.

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

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

8.5 Conditional expressions

***ifExpression*:**
 `'if'`
 {*NL*}
 `'('`
 {*NL*}
 expression
 {*NL*}
 `')'`
 {*NL*}
 (*controlStructureBody* | ([*controlStructureBody*] {*NL*} [`';'`] {*NL*} `'else'` {*NL*} (*controlStructureBody* | `';'`)) | `';'`)

Conditional expressions use a boolean value of one expression (*condition*) to decide which of the two control structure bodies (*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 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` and may be used only as a statement.

> Example:
>
> ```
> // x has type kotlin.Int and value 1
> val x = if (true) 1 else 2
> // illegal, as if expression without false branch
> //   cannot be used as an expression
> val y = if (true) 1
> ```

The type of the condition expression must be a subtype of `kotlin.Boolean`, otherwise it is a compile-time 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.

Example:

```
x = if (true) 1 else 2
```

is the same as

```
x = (if (true) 1 else 2)
```

At the same time

```
if (true) x = 1 else x = 2
```

is the same as

```
if (true) (x = 1) else (x = 2)
```

8.6 When expressions

***whenExpression*:**
`'when'`
{*NL*}
[*whenSubject*]
{*NL*}
`'{'`
{*NL*}
{*whenEntry* {*NL*}}
{*NL*}
`'}'`

***whenEntry*:**
(*whenCondition* {{*NL*} `','` {*NL*} *whenCondition*} [{*NL*} `','`] {*NL*} `'->'` {*NL*} *controlStructureBody* [*semi*])
| (`'else'` {*NL*} `'->'` {*NL*} *controlStructureBody* [*semi*])

***whenCondition*:**
expression
| *rangeTest*
| *typeTest*

***rangeTest*:**
inOperator {*NL*} *expression*

***typeTest*:**
isOperator {*NL*} *type*

When expression is similar to a conditional expression in that it allows one of several different control structure bodies (*cases*) to be evaluated, depending on some boolean conditions. The key difference is 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 changes to the result of when expression.

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

- *Type test condition*: type checking operator followed by a type (`is T` or `!is T`). The resulting condition is a type check expression of the form `boundValue is T` or `boundValue !is T`.
- *Contains test condition*: containment operator followed by an expression (`in Expr` or `!in Expr`). The resulting condition is a containment check expression of the form `boundValue in Expr` or `boundValue !in Expr`.
- *Any other applicable 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.

> Note: in Kotlin version 1.3 and earlier, simple (unlabeled) `break` and `continue` expressions were disallowed in when expressions.

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

Examples:

```
val a = 42
val b = -1

when {
  a == b -> {}
  a != b -> {}
}

// Error, as it is a non-exhaustive when expression
val c = when {
  a == b -> {}
  a != b -> {}
}

val d = when {
  a == b -> {}
  a != b -> {}
  else -> {}
}

when {
  a == b || a != b -> {}
  42 > 0 -> {}
}

val a = 42
val b = -1

val l = (1..10).toList()

when (a) {
  is Int, !is Int -> {}
  in l, !in l -> {}
}

// Error, as it is a non-exhaustive when expression
val c = when (a) {
  is Int, !is Int -> {}
  in l, !in l -> {}
}
```

```
val d = when (a) {
  is Int, !is Int -> {}
  in l, !in l -> {}
  else -> {}
}
```

When with bound value also allows for an in-place property declaration of the form `when (val V = E) { ... }` inside the parentheses. This declares a new property (see declaration sections for details) alongside the usual mechanics of the *when-expression*. The scope of this property is limited to the `when` expression, including both conditions and control structure bodies of the expression. As its form is limited to a simple "assignment-like" declaration with an initializer, this property does not allow getters, setters, delegation or destructuring. It is also required to be immutable. Conceptually, it is very similar to declaring such a property before the when-expression and using it as subject, but with a difference in scoping of this property described above.

Example:

```
when(val a = b + c) {
    !is Foo -> a + 1
    else -> b
}

val y = a // illegal, a is not visible here anymore
```

8.6.1 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 evaluating to `true`;
    - A constant expression evaluating to `false`;
  - The bound expression is of a `sealed` class or interface `S` and all of its *direct non-sealed subtypes* $T_1, \ldots, T_n$ are covered in this expression. A subtype T_i is considered covered if when expression contains one of the following:
    - a type test condition `is` S_j, where $S_j <: S, T_i <: S_j$;
    - a type test condition `!is` S_j, where $S_j <: S, T_i \not<: S_j, \exists k \neq i : T_k <: S_j$.
      Note: in case the set of direct non-sealed subtypes for sealed type `S` is empty (i.e., its sealed hierarchy is uninhabited), the exhaustiveness of when expression is implementation-defined.

    Additionally, an enum subtype E_i is considered covered also if all its enumerated values are checked for equality using constant expression;

  – The bound expression is of an `enum class` type and all its enumerated values are checked for equality using constant expression;
  – The bound expression is of a nullable type *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`.

For object types, the type test condition may be replaced with equality check with the object value.

> Note: if one were to override `equals` for an object type incorrectly (i.e., so that an object is not equal to itself), it would break the exhaustiveness check. It is unspecified whether this situation leads to an exception or an undefined value for this `when` expression.

```
sealed class Base
class Derived1: Base()
object Derived2: Base()

val b: Base = ...

val c = when(b) {
    is Derived1 -> ...
    Derived2 -> ...
    // no else needed here
}

sealed interface I1
sealed interface I2
sealed interface I3

class D1 : I1, I2
class D2 : I1, I3

sealed class D3 : I1, I3

fun foo() {
    val b: I1 = mk()

    val c = when(a) {
        !is I3 -> {} // covers D1
        is D2 -> {} // covers D2
        // D3 is sealed and does not take part
        // in the exhaustiveness check
    }
}
```

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

8.7 Logical disjunction expressions

***disjunction*:**
  conjunction {{*NL*} `'||'` {*NL*} *conjunction*}

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 compile-time error. The type of logical disjunction expression is `kotlin.Boolean`.

8.8 Logical conjunction expressions

***conjunction*:**
  equality {{*NL*} `'&&'` {*NL*} *equality*}

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 compile-time error. The type of logical disjunction expression is `kotlin.Boolean`.

8.9 Equality expressions

***equality*:**
  comparison {*equalityOperator* {*NL*} *comparison*}

***equalityOperator*:**
  `'!='`
  | `'!=='`
  | `'=='`
  | `'==='`

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

8.9.1 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. In particular, this means that two values acquired by the same constructor call are equal by reference, while two values created by two different constructor calls are not equal by reference. A value created by any constructor call is never equal by reference to a null reference.

There is an exception to these rules: values of value classes are not guaranteed to be reference equal even if they are created by the same constructor invocation as said constructor invocation is explicitly allowed to be inlined by the compiler. It is thus highly discouraged to compare value classes by reference.

For special values created without explicit constructor calls, notably, constant literals and constant expressions composed of those literals, and for values of value classes, 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 should not be used as a means of comparing such values.

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

Kotlin checks the applicability of reference equality operators at compile-time and may reject certain combinations of types for `A` and `B`. Specifically, it uses the following basic principle.

> If type of `A` and type of `B` are definitely distinct and not related by subtyping, `A === B` is an invalid expression and should result in a compile-time error.
>
> Informally: this principle means "no two objects of different types can be equal by reference".

8.9.2 Value equality expressions

Value equality expressions are binary expressions which use value equality operators: `==` and `!=`. These operators are overloadable, but are different from other overloadable operators in that the expansion depends on the form of the arguments.

Reference equality contract for the `equals` method implementation consists of the following requirements imposed on `kotlin.Any.equals` override:

1. $\forall \mathtt{A}, \mathtt{B} : \mathtt{A} === \mathtt{B} \implies \mathtt{A.equals(B)}$
2. $\forall \mathtt{A}, \mathtt{B} : \mathtt{B} === \mathtt{null} \implies \mathtt{!A.equals(B)}$

The operators themselves have the following expansion:

- `A != B` is exactly the same as `!(A == B)`;
- `A == B` has a more complex expansion:

- If either of `A` or `B` is a null literal `null`, then `A == B` is exactly the same as `A === B`;
- If both of `A` and `B` have compile-time types that are built-in floating point arithmetic types or their nullable variants, then `A == B` is exactly the same as `(A === null && B === null) || (A !== null && B !== null && ieee754Equals(A!!, B!!))` where `ieee754Equals` is a special intrinsic function unavailable in user-side Kotlin which performs equality comparison of two floating-point numbers according to IEEE 754 equality specification;
- Otherwise, `A == B` is semantically equivalent to `(A as? Any)?.equals(B as Any?) ?: (B === null)`, assuming that `operator equals` abides the reference equality contract. This means that if the compiler implementation can prove that `A === null` or `B === null` or `A === B`, this expansion may be optimized to never call `equals` function at all.

> Note: the expansion involving a call to `equals` operator function always resolves to the member function of `kotlin.Any` as there is no way to provide a more suitable overload candidate. Furthermore, it is not possible to write an `operator`-qualified function with this name that is not an override of this member function.

> Note: the floating-point type expansion given above means that, in some situations and on some platforms, `A == B` and `(A as Any?) == (B as Any?)` may produce different results if `A` and `B` are floating-point numbers. For example, on JVM platform the overridden `equals` implementation for floating-point numbers does not follow the IEEE 754 definition of equality, so `A == A` is false, while `(A as Any?) == (A as Any?)` is true if `A` has a `NaN` value.

Value equality expressions always have type `kotlin.Boolean` as does the `equals` method in `kotlin.Any`.

Kotlin checks the applicability of value equality operators at compile-time and may reject certain combinations of types for `A` and `B`. Specifically, it uses the following basic principle.

> If type of `A` and type of `B` are definitely distinct and not related by subtyping, `A == B` is an invalid expression and should result in a compile-time error.

> Informally: this principle means "no two objects unrelated by subtyping can ever be considered equal by `==`".

8.10 Comparison expressions

***comparison*:**

genericCallLikeComparison {*comparisonOperator* {*NL*} *genericCallLike-Comparison*}

***comparisonOperator*:**
`'<'`
| `'>'`
| `'<='`
| `'>='`

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

- If both `A` and `B` have the same compile-time type which is also one of the built-in floating point arithmetic types, then:
  - `A < B` is exactly the same as `ieee754Less(A, B)`
  - `A > B` is exactly the same as `ieee754Less(B, A)`
  - `A <= B` is exactly the same as `ieee754Less(A, B) || ieee754Equals(A, B)`
  - `A >= B` is exactly the same as `ieee754Less(B, A) || ieee754Equals(A, B)`
- Otherwise:
  - `A < B` is exactly the same as `integerLess(A.compareTo(B), 0)`
  - `A > B` is exactly the same as `integerLess(0, A.compareTo(B))`
  - `A <= B` is exactly the same as `!integerLess(0, A.compareTo(B))`
  - `A >= B` is exactly the same as `!integerLess(A.compareTo(B), 0)`

where `compareTo` is a valid operator function available in the current scope, `integerLess` is a special intrinsic function unavailable in user-side Kotlin which performs integer "less-than" comparison of two integer numbers and `ieee754Less` and `ieee754Equals` are special intrinsic functions unavailable in user-side Kotlin which perform IEEE 754 compliant "less-than" and equality comparison respectively.

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

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

8.11 Type-checking and containment-checking expressions

***infixOperation*:**
elvisExpression {(*inOperator* {*NL*} *elvisExpression*) | (*isOperator* {*NL*} *type*)}

***inOperator*:**
`'in'`

| *NOT_IN*

***isOperator*:**
`'is'`
| *NOT_IS*

8.11.1 Type-checking expressions

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. 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.

The type T must be runtime-available, otherwise it is a compile-time error.

If the type T is not a parameterized type, it must be runtime-available, otherwise it is a compile-time error. If T is a parameterized type, the bare type argument inference is performed for the compile-time known type of E and the type constructor TC of T. After that, given the result arguments of this bare type inference $A_0, A_1 \ldots A_N$, T must suffice the constraint $T!! <: TC[A_0, A_1 \ldots A_N]$, checking each of its argument for conformance with the type of E.

Example:

```
interface Foo<A, B>
class Fee<T, U>: Foo<U, T>

fun f(foo: Foo<String, Int>) {
    // valid: you can specify parameters
    // as long as they correspond to base type
    if(foo is Fee<Int, String>) { ... }
    // invalid: Fee<String, Int> is not a subtype
    // of Foo<String, Int>
    if(foo is Fee<String, Int>) { ... }
    // valid: may be specified partially
    if(foo is Fee<Int, *>) { ... }
```

T may also be specified without arguments by using the *bare type syntax* in which case the same process of bare type argument inference is performed, with the difference being that the resulting arguments $A_0, A_1 \ldots A_N$ are used as arguments for T directly. If any of these arguments are inferred to be star-projections, this is a compile-time error.

Example:

```
interface Foo<A, B>
class Fee<T, U>: Foo<U, T>

fun f(foo: Foo<String, Int>) {
```

```
// valid: same as foo is Fee<Int, String>
if(foo is Fee) { ... }
```

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, for further details, refer to the corresponding section.

8.11.2 Containment-checking expressions

A *containment-checking expression* is a binary expression which uses a containment operator `in` or `!in`. These operators are overloadable 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`.

8.12 Elvis operator expressions

***elvisExpression*:**
  infixFunctionCall {{*NL*} *elvis* {*NL*} *infixFunctionCall*}

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 of the non-nullable variant of the type of the left-hand side expression and the type of the right-hand side expression.

8.13 Range expressions

***rangeExpression*:**
additiveExpression {(`'..'` | `'..<'`) {*NL*} *additiveExpression*}

A *range expression* is a binary expression which uses a range operator `..` or a range-until operator `..<`. These are overloadable operators with the following expansions:

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

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

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

8.14 Additive expressions

***additiveExpression*:**
multiplicativeExpression {*additiveOperator* {*NL*} *multiplicativeExpression*}

***additiveOperator*:**
`'+'`
| `'-'`

An *additive expression* is a binary expression which uses an addition (`+`) or subtraction (`-`) operators. These are overloadable 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.

8.15 Multiplicative expressions

***multiplicativeExpression*:**
asExpression {*multiplicativeOperator* {*NL*} *asExpression*}

***multiplicativeOperator*:**
`'*'`

| `'/'`
| `'%'`

A *multiplicative expression* is a binary expression which uses a multiplication (`*`), division (`/`) or remainder (`%`) operators. These are overloadable 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: in Kotlin version 1.3 and earlier, there was an additional overloadable operator for `%` called `mod`, which has been removed in Kotlin 1.4.

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.

8.16 Cast expressions

***asExpression*:**
prefixUnaryExpression {{*NL*} *asOperator* {*NL*} *type*}

***asOperator*:**
`'as'`
| `'as?'`

A *cast expression* is a binary expression which uses 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 *an unchecked cast* expression. This expression perform a runtime check whether the runtime type of E is a subtype of T and throws an exception otherwise. If type T is a runtime-available type without generic parameters, then this exception is thrown immediately when evaluating the cast expression, otherwise it is implementation-defined whether an exception is thrown at this point.

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 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.

If type T is a runtime-available type **with** generic parameters, type parameters are **not** checked w.r.t. subtyping. This is another potentially erroneous situation, which should be reported as a compile-time warning.

Similarly to type checking expressions, some type arguments may be excluded from this check if they are known from the supertype of E and, if all type arguments of T can be inferred, they may be omitted altogether using the *bare type syntax*. See type checking section for explanation.

The checked cast expression result has the type which is the nullable variant of the type T specified in the expression.

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

8.17 Prefix expressions

***prefixUnaryExpression*:**
  {*unaryPrefix*} *postfixUnaryExpression*

***unaryPrefix*:**
  annotation
  | *label*
  | (*prefixUnaryOperator* {*NL*})

***prefixUnaryOperator*:**
  `'++'`
  | `'--'`
  | `'-'`
  | `'+'`
  | *excl*

8.17.1 Annotated expressions

Any expression in Kotlin may be prefixed with any number of annotations. These do not change the value of the expression and can be used by external tools and for implementing platform-dependent features. See annotations chapter of this document for further information and examples of annotations.

8.17.2 Prefix increment expressions

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

- `++A` is exactly the same as `when(val $tmp = A.inc()) { else -> A = $tmp; $tmp }` where `inc` is a valid operator function available in the current scope.

Informally: `++A` assigns the result of `A.inc()` to `A` and also returns it as the result.

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

As the result of `inc` is assigned to `A`, the return type of `inc` must be a subtype of `A`. Otherwise, such declaration is a compile-time error.

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

8.17.3 Prefix decrement expressions

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

- `--A` is exactly the same as `when(val $tmp = A.dec()) { else -> A = $tmp; $tmp }` where `dec` is a valid operator function available in the current scope.

  Informally: `--A` assigns the result of `A.dec()` to `A` and also returns it as the result.

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

As the result of `dec` is assigned to `A`, the return type of `dec` must be a subtype of `A`. Otherwise, such declaration is a compile-time error.

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

8.17.4 Unary minus expressions

An *unary minus* expression is an expression which uses the prefix form of operator `-`. It is an overloadable 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.

8.17.5 Unary plus expressions

An *unary plus* expression is an expression which uses the prefix form of operator `+`. It is an overloadable 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.

8.17.6 Logical not expressions

A *logical not* expression is an expression which uses the prefix operator `!`. It is an overloadable 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.

8.18 Postfix operator expressions

***postfixUnaryExpression*:**
  primaryExpression {*postfixUnarySuffix*}

***postfixUnarySuffix*:**
  postfixUnaryOperator
  | *typeArguments*
  | *callSuffix*
  | *indexingSuffix*
  | *navigationSuffix*

***postfixUnaryOperator*:**
  `'++'`
  | `'--'`
  | (`'!'` *excl*)

8.18.1 Postfix increment expressions

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

- `A++` is exactly the same as `when(val $tmp = A) { else -> A = $tmp.inc(); $tmp }` where `inc` is a valid operator function available in the current scope.

  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. Otherwise, it is a compile-time error.

As the result of `inc` is assigned to `A`, the return type of `inc` must be a subtype of `A`. Otherwise, such declaration is a compile-time error.

A postfix increment expression has the same type as its operand expression (for our examples, the type of `A`).

8.18.2 Postfix decrement expressions

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

- `A--` is exactly the same as `when(val $tmp = A) { else -> A = $tmp.dec(); $tmp }` where `dec` is a valid operator function available in the current scope.

  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. Otherwise, it is a compile-time error.

As the result of `dec` is assigned to `A`, the return type of `dec` must be a subtype of `A`. Otherwise, such declaration is a compile-time error.

A postfix decrement expression has the same type as its operand expression (for our examples, the type of `A`).

8.19 Not-null assertion expressions

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 not-null assertion expression is the non-nullable variant of the type of `e`.

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

8.20 Indexing expressions

***postfixUnaryExpression*:**
  primaryExpression {*postfixUnarySuffix*}

***postfixUnarySuffix*:**
  postfixUnaryOperator
  | *typeArguments*
  | *callSuffix*
  | *indexingSuffix*
  | *navigationSuffix*

***indexingSuffix*:**
  `'['`
  {*NL*}
  expression
  {{*NL*} `','` {*NL*} *expression*}
  [{*NL*} `','`]
  {*NL*}
  `']'`

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

It is an overloadable 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, for a corresponding assignment form, see here.

8.21 Call and property access expressions

***postfixUnaryExpression*:**
  primaryExpression {*postfixUnarySuffix*}

***postfixUnarySuffix*:**
  postfixUnaryOperator
  | *typeArguments*
  | *callSuffix*
  | *indexingSuffix*
  | *navigationSuffix*

***navigationSuffix*:**
  memberAccessOperator {*NL*} (*simpleIdentifier* | *parenthesizedExpression* | `'class'`)

***callSuffix*:**
  [*typeArguments*] (([*valueArguments*] *annotatedLambda*) | *valueArguments*)

***annotatedLambda*:**
  {*annotation*} [*label*] {*NL*} *lambdaLiteral*

***valueArguments*:**
  `'('` {*NL*} [*valueArgument* {{*NL*} `','` {*NL*} *valueArgument*} [{*NL*} `','`] {*NL*}] `')'`

***typeArguments*:**
  `'<'`

{*NL*}
typeProjection
{{*NL*} `','` {*NL*} *typeProjection*}
[{*NL*} `',`']
{*NL*}
`'>'`

***typeProjection*:**
([*typeProjectionModifiers*] *type*)
| `'*'`

***typeProjectionModifiers*:**
typeProjectionModifier {*typeProjectionModifier*}

***memberAccessOperator*:**
({*NL*} `'.'`)
| ({*NL*} *safeNav*)
| `'::'`

8.21.1 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 value available in the current scope, while the right side corresponds to a declaration in the scope of that value.

  Note: qualification uses operator `.` only.

- A property access. Here `a` is a value available in the current scope and `c` is a property name.

  Note: the navigation operator `.` is closely related to the concept of paths.

If followed by the call suffix (arguments in parentheses), `a.c()` may have one of the following semantics when used as an expression:

- A function call; here `a` is a value available in the current scope and `c` is a function name;
- A property access with `invoke`-convention; here `a` is a value available in the current scope and `c` is a property name.

  These expressions follow the overloading 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

  ```
  when (val $tmp = a) {
      null -> null
      else -> { $tmp.c }
  }
  ```

  for any right-hand combinations of operators present in `c`, which are expanded further, as usual.

The type of `a?.c` is the nullable variant of the type of `a.c`.

> Note: safe navigation expression may also include the call suffix as `a?.c()`and is expanded in a similar fashion.

8.21.2 Callable references

Callable references are a special kind of expressions used to refer to callables (properties and functions) without actually calling/accessing them. They are not to be confused with class literals which use similar syntax, but with the keyword `class` instead of an 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 type `A`. A callable reference `e::c` where `e` is an expression of type `E` and `c` is a name of a callable available for type `E` is a *callable reference* for expression `e`. The exact callable selected when using this syntax is based on overload resolution much like when accessing the value of a property using the `.` navigation operator. However, in some cases there are important differences which we cover in the corresponding paragraphs.

Depending on the meaning of the left-hand and right-hand sides of a callable reference `lhs::rhs`, the value of the whole expression is defined as follows.

- If `lhs` is a type, but not a value (an example of a type which can also be used as a value is an object type), while `rhs` is resolved to refer to a property of `lhs`, `lhs::rhs` is a *type-property* reference;
- If `lhs` is a type, but not a value (an example of a type which can also be used as a value is an object type), while `rhs` is resolved to refer to a function available on `rhs`, `lhs::rhs` is a *type-function* reference;
- If `lhs` is a value, while `rhs` is resolved to refer to a property of `lhs`, `lhs::rhs` is a *value-property* reference;
- If `lhs` is a value, while `rhs` is resolved to refer to a function available on `rhs`, `lhs::rhs` is a *value-function* reference.

Important: callable references to callables which are a member and an extension (that is, an extension to one type declared as a member of a classifier) are forbidden

Examples:

```
class A {
    val a: Int = 42

    fun a(): String = "TODO()"

    companion object {
        val bProp: Int = 42

        fun bFun(): String = "TODO()"
    }
}

object O {
    val a: Int = 42

    fun a(): String = "TODO()"
}

fun main() {
    // Error: ambiguity between two possible callables
    // val errorAmbiguity = A::a

    // Error: cannot reference companion object implicitly
    // val errorCompanion = A::bFun

    val aTypePropRef: (A) -> Int = A::a

    val aTypeFunRef: (A) -> String = A::a

    val aValPropRef: () -> Int = A()::a

    val aValFunRef: () -> String = A()::a

    // Error: object type behave as values
    // val oTypePropRef: (O) -> Int = O::a

    // Error: object types behave as values
    // val oTypeFunRef: (O) -> String = O::a

    val oValPropRef: () -> Int = O::a
```

```
        val oValFunRef: () -> String = O::a
    }
```

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

- The type of any 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 function reference is a subtype of `kotlin.reflect.KFunction<T>`, where the type parameter `T` is fixed to the return type of the function;
- The type of any callable reference is a subtype of function type which allows the corresponding callable to be accessed/called accordingly.
  - For a type-callable reference `lhs::rhs`, it is a function type `(O, Arg0 ... ArgN) -> R`, where `O` is a receiver type (type of `lhs`), `Arg0, ... , ArgN` are either empty (for a property reference) or the types of function formal parameters (for a function reference), and `R` is the result type of the callable;
  - For a value-callable reference `lhs::rhs`, it is a function type `(Arg0 ... ArgN) -> R`, where `Arg0, ... , ArgN` are either empty (for a property reference) or the types of function formal parameters (for a function reference), and `R` is the result type of the callable. The receiver of such callable reference is bound to `lhs`.

Being of a function type also means callable references are valid callables themselves, with an appropriate operator `invoke` overload, which allows using call syntax to evaluate such callable with the suitable arguments.

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

Please note that the above holds for *resolved* callable references, where it is known what entity a particular reference references. In the general case, however, it is unknown as the overload resolution must be performed first. Please refer to the corresponding section for details.

8.21.3 Class literals

A class literal is similar in syntax to a callable reference, with the difference being that it uses the keyword `class`. Similar to callable references, there are two forms of class literals: type and value class literals.

> Note: class literals are one of the few cases where a parameterized type may (and actually **must**) be used without its type parameters.

All class literals `lhs::class` are of type `kotlin.KClass<T>` and produce a platform-defined object associated with type `T`, which, in turn, is either the `lhs` type or the runtime type of the `lhs` value. In both cases, `T` must be a runtime-available non-nullable type. As the runtime type of any expression cannot be known at compile time, the compile-time type of a class literal is `kotlin.KClass<U>` where $T <: U$ and `U` is the compile-time type of `lhs`.

A class literal can be used to access platform-specific capabilities of the runtime type information available on the current platform, either directly or through reflection facilities.

8.21.4 Function calls and property access

Function call expression is an expression used to invoke functions. Property access expression is an expression used to access properties.

There are two kinds of both: with and without explicit receiver (the left-hand side of the `.` operator). For details on how a particular candidate and receiver for a particular call / property access is chosen, please refer to the Overload resolution section.

> Important: in some cases function calls are syntactically indistinguishable from property accesses with `invoke`-convention call suffix.

From this point on in this section we well refer to both as function calls. As described in the function declaration section, function calls receive arguments of several different kinds:

- Explicit receiver argument, used in calls with explicit receivers;
- Normal arguments, provided directly inside the parentheses part of the call;
- Named arguments in the form `identifier = value`, where `identifier` is a parameter name used at declaration-site of the function;
- Variable length arguments, provided the same way as normal arguments;
- A trailing lambda literal argument, specified outside the parentheses (see lambda literal section for details).

In addition to these, a function declaration may specify a number of default parameters, which allow one to omit specifying them at call-site, in which case their default value is used during the evaluation.

The evaluation of a function call begins with the evaluation of its explicit receiver, if it is present. Function arguments are then evaluated **in the order of their appearance in the function call** left-to-right, with no consideration on how the parameters of the function were specified during function declaration. This means that, even if the order at declaration-site was different, arguments at call-site are evaluated in the order they are given. Default arguments not specified in the call are all evaluated **after** all provided arguments, in the order of their appearance in function declaration. Afterwards, the function itself is invoked.

> Note: this means default argument expressions which are used (i.e., for which the call-site does not provide explicit arguments) are reevaluated at every such call-site. Default argument expressions which

are not used (i.e., for which the call-site provides explicit arguments) are **not** evaluated at such call-sites.

Examples: we use a notation similar to the control-flow section to illustrate the evaluation order.

```
fun f(x: Int = h(), y: Int = g())
...
f() // $1 = h(); $2 = g(); $result = f($1, $2)
f(m(), n()) // $1 = m(); $2 = n(); $result = f($1, $2)
f(y = n(), x = m()) // $1 = n(); $2 = m(); $result = f($2, $1)
f(y = n()) // $1 = n(); $2 = h(); $result = f($2, $1)
```

```
fun f(x: Int = h(), y: () -> Int)
...
f(y = {2}) // $1 = {2}; $2 = h(); $result = f($2, $1)
f { 2 } // $1 = {2}; $2 = h(); $result = f($2, $1)
f(m()) { 2 } // $1 = m(); $2 = {2}; $result = f($1, $2)
```

Operator calls work in a similar way: every operator evaluates in the same order as its expansion does, unless specified otherwise.

Note: this means that the containment-checking operators are effectively evaluated right-to-left w.r.t. their expansion.

8.21.5 Spread operator expressions

***postfixUnaryExpression*:**
primaryExpression {*postfixUnarySuffix*}

Spread operator expression is a special kind of expression which is only applicable in the context of calling a function with variable length parameters. For a spread operator expression `*E` it is required that `E` is of an array type and the expression itself is used as a value argument to a function call. This allows passing an array as a *spread* value argument, providing the elements of an array as the variable length argument of a callable. It is allowed to mix spread arguments with regular arguments, all fitting into the same variable length argument slot, with elements of all spread arguments supplied in sequence.

Example:

```
fun foo(vararg c: String) { ... }
...
val a: String = "a"
val b: Array<String> = arrayOf("b", "c", "d")
val c: String = "e"
val d: Array<String> = arrayOf()
val e: Array<String> = arrayOf("f", "g")
...
foo(a, *b, c, *d, *e)
```

```
// is equivalent to
foo("a", "b", "c", "d", "e", "f", "g")
```

Spread operator expressions are not allowed in any other context. See Variable length parameter section for details.

The type of a spread argument must be a subtype of ATS(`kotlin.Array(out` T)) for a variable length parameter of type T.

> Example: for parameter `vararg a: Int` the type of a corresponding spread argument must be a subtype of `IntArray`, for parameter `vararg b: T` where `T` is a classifier type the type of a corresponding spread argument must be a subtype of `Array<out T>`.

8.22 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. However, sometimes it does not make much sense to provide a separate function declaration, when one would 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 a number of differences which we discuss in their respective sections.

8.22.1 Anonymous function declarations

***anonymousFunction*:**
  ['suspend']
  {*NL*}
  'fun'
  [{*NL*} *type* {*NL*} '.']
  {*NL*}
  parametersWithOptionalType
  [{*NL*} ':' {*NL*} *type*]
  [{*NL*} *typeConstraints*]
  [{*NL*} *functionBody*]

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 cannot have type parameters;
- Anonymous functions cannot have default parameters;
- Anonymous functions may have variable length parameters, but they are automatically decayed to non-variable length parameters of the corresponding

array type via array type specialization;
- Anonymous functions may omit formal parameter types and return type, if they can be inferred from the context.

Anonymous function declaration can declare an anonymous extension function by following the extension function declaration convention.

> 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.

8.22.2 Lambda literals

***lambdaLiteral*:**
  `'{'`
  {*NL*}
  [[*lambdaParameters*] {*NL*} `'->'` {*NL*}]
  statements
  {*NL*}
  `'}'`

***lambdaParameters*:**
  lambdaParameter {{*NL*} `','` {*NL*} *lambdaParameter*} [{*NL*} `','`]

***lambdaParameter*:**
  variableDeclaration
  | (*multiVariableDeclaration* [{*NL*} `':'` {*NL*} *type*])

Lambda literals are similar to anonymous function declarations in that they define a function with no name. Unlike them, however, lambdas 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 have the same restrictions as anonymous function declarations, but additionally cannot have `vararg` parameters.

Lambda literals can introduce destructuring parameters. Lambda parameter of the form `(a, b, ..., n)` (note the parenthesis) declares a destructuring formal parameter, which references the actual argument and its `componentN()` functions as follows (see the operator overloading section for details).

```
val plus: (Pair<Int, Double>) -> String = { (i, d) ->
    "$i + $d = ${i + d}"
}

val plus: (Pair<Int, Double>) -> String = { p ->
    val i = p.component1()
    val d = p.component2()
    "$i + $d = ${i + d}"
}
```

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

If a lambda expression has no explicit parameter list, but does have one parameter, this parameter can be accessed inside the lambda body using a special property called `it`.

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

Any lambda may define either a normal function or an extension function, the exact case dependent on the use context of the lambda. If a lambda expression defines an extension function, its extension receiver may be accessed using the standard `this` syntax inside the lambda body.

Lambda literals are different from other forms of function declarations in that non-labeled `return` expressions inside lambda body refer to the outer non-lambda function the expression is used in rather than the lambda expression itself. Such non-labeled returns are only allowed if the lambda and all its parent lambdas (if present) are guaranteed to be inlined, otherwise it is a compile-time error.

If a lambda expression is labeled, it can be returned from using a labeled return expression.

If a **non-labeled** lambda expression is used as a parameter to a function call, the name of the function called may be used as a label.

If a labeled `return` expression is used when there are several matching labels available (e.g., inside several nested function calls with the same name), this is resolved as `return` to the nearest matching label.

> Example:
>
> ```
> // kotlin.run is a standard library inline function
> //   receiving a lambda parameter
>
> fun foo() { // (1)
>     run b@ { // (2)
>         run b@ { // (3)
>             return; // returns from (1)
> ```

```
        }
    }
}

fun bar() { // (1)
    run b@ { // (2)
        run b@ { // (3)
            return@b; // returns from (3)
        }
    }
}

fun baz() { // (1)
    run b@ { // (2)
        run c@ { // (3)
            return@b; // returns from (2)
        }
    }
}

fun qux() { // (1)
    run { // (2)
        run { // (3)
            return@run; // returns from (3)
        }
    }
}

fun quux() { // (1)
    run { // (2)
        run b@ { // (3)
            return@run; // returns from (2)
        }
    }
}

fun quz() { // (1)
    run b@ { // (2)
        run b@ { // (3)
            return@run; // illegal: both run invocations are labeled
        }
    }
}
```

Any properties used inside the lambda body are **captured** by the lambda expression and, depending on whether it is inlined or not, affect how these properties are processed by other mechanisms, e.g. smart casts. See corresponding sections for details.

8.23 Object literals

***objectLiteral*:**
    [`'data'`]
    {*NL*}
    `'object'`
    [{*NL*} `':'` {*NL*} *delegationSpecifiers* {*NL*}]
    [{*NL*} *classBody*]

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.

> Note: in object literals, only inner classes are allowed; interfaces, objects or nested classes are forbidden.

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 a 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 declaring scope, has 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 type inference.

> Note: in this context "escaping current scope" is performed immediately if the corresponding value is declared as a **non-private** global- or classifier-scope property, as those are parts of an externally accessible interface.

> Example:

```
open class Base
interface I

class M {
        fun bar() = object : Base(), I {}
        // Error, as public return type of `bar`
        //   cannot be anonymous

        fun baz(): Base = object : Base(), I {}
        // OK, as an anonymous type is implicitly
        //   cast to Base

        private fun qux() = object : Base(), I {}
        // OK, as an anonymous type does not escape
        //   via private functions

        private fun foo() = object {
            fun bar() { println("foo.bar") }
        }

        fun test1() = foo().bar()

        fun test2() = foo()
        // OK, as an anonymous type is implicitly
        //   cast to Any
    }

    fun main() {
        M().test1() // OK
        M().test2().bar() // Error: Unresolved reference: bar
    }
```

8.23.1 Functional interface lambda literals

If a lambda literal is preceded with a functional interface name, this expression defines an anonymous object, implementing the specified functional interface via the provided lambda literal (which becomes the implementation of its single abstract method).

To be a well-formed functional interface lambda literal, the type of lambda literal must be a subtype of the associated function type of the specified functional interface.

8.24 This-expressions

***thisExpression*:**
    `'this'`
    | *THIS_AT*

This-expressions are special kind of expressions used to access receivers available in the current scope. The basic form of this expression, denoted by a non-labeled `this` keyword, is used to access the default implicit receiver according to the receiver priority. In order to access other implicit 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 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;
- `this@outerFunction`, where `outerFunction` is the name of a function which takes lambda literal currently being declared as an immediate argument (that is, this-expression is located in the lambda expression body), refers to the implicit receiver object of the lambda expression.

  Note: `this@outerFunction` notation is mutually exclusive with `this@lambda` notation, meaning if a lambda literal is labeled `this@outerFunction` cannot be used.

  Note: `this@outerFunction` and `this@label` notations can be used only in lambda literals which have an extension function type, i.e., have an implicit receiver.

  Important: any other forms of this-expression are illegal and should result in a compile-time error.

In case there are several entities with the same label, labeled `this` refers to the closest label.

Example:

```
interface B
object C

class A/* receiver (1) */ {
    fun B/* receiver (2) */.foo() {
        // `run` is a standard library function
        //   with an extension lambda parameter
        C/* receiver (3) */.run {
            this // refers to receiver (3) of type C
            this@A // refers to receiver (1) of type A
            // this@B // illegal: B is not being declared
            this@foo // refers to receiver (2) of type B
```

```
                this@run // refers to receiver (3) of type C
            }
            C/* receiver (4) */.run label@{
                this // refers to receiver (4) of type C
                this@A // refers to receiver (1) of type A
                // this@B // illegal: B is not being declared
                this@foo // refers to receiver (2) of type B
                this@label // refers to receiver (4) of type C
                // this@run // illegal: lambda literal is labeled
            }
        }
    }
```

8.25 Super-forms

***superExpression*:**
    (`'super'` [`'<'` {*NL*} *type* {*NL*} `'>'`] [*AT_NO_WS simpleIdentifier*])
    | *SUPER_AT*

Super-forms are special kind of expression which can only be used as receivers in a call 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 implementations from the immediate supertypes without invoking overriding behaviour.

If an implementation is not available (e.g., one attempts to access an abstract method of a supertype in this fashion), this is a compile-time error.

The basic form of this expression, denoted by `super` keyword, is used to access the immediate supertype of the currently declared classifier selected as a part of overload resolution. In order to access a specific supertype implementations, extended `super` expressions are used. These may be any of the following:

- `super<Klazz>`, where `Klazz` is a name of one of the immediate supertypes of the currently declared classifier, refers to that supertype and its implementations;
- `super<Klazz>@type`, where `type` is a name of any currently declared classifier and `Klazz` is a name of one of the immediate supertypes of the `type` classifier, refers to that supertype and its implementations.

  > Note: `super<Klazz>@type` notation can be used only in inner classes, as only inner classes can have access to supertypes of other classes, i.e., supertypes of their parent class.

  Example:

  ```
  interface A {
  ```

```
    fun foo() { println("A") }
}
interface B {
    fun foo() { println("B") }
}

open class C : A {
    override fun foo() { println("C") }
}

class E : C() {
    init {
        super.foo() // "C"
        super<C>.foo() // "C"
    }
}

class D : C(), A, B {
    init {
        // Error: ambiguity as several immediate supertypes
        //   with callable `foo` are available here
        // super.foo()
        super<C>.foo() // "C"
        super<B>.foo() // "B"
        // Error: A is *not* an immediate supertype,
        //   as C inherits from A and is considered
        //   to be "more immediate"
        // super<A>.foo()
    }

    inner class Inner {
        init {
            // Error: C is not available
            // super<C>.foo()
            super<C>@D.foo() // "C"
            super<B>@D.foo() // "B"
        }
    }

    override fun foo() { println("D") }
}
```

8.26 Jump expressions

***jumpExpression*:**

('throw' {*NL*} *expression*)
| (('return' | *RETURN_AT*) [*expression*])
| 'continue'
| *CONTINUE_AT*
| 'break'
| *BREAK_AT*

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`, meaning that they never produce any runtime value;
- Any code which follows such expressions is never evaluated.

8.26.1 Throw expressions

Throw expression `throw e` allows throwing exception objects. A valid throw expression `throw e` requires that:

- `e` is a value of a runtime-available type;
- `e` is a value of an exception type.

Throwing an exception results in checking active `try`-blocks. See the Exceptions section for details.

8.26.2 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 non-labeled `return` keyword, which returns from the innermost function declaration (or anonymous function declaration), and a labeled return expression of the form `return@Context` which works as follows.

- If `return@Context` is used inside a named function declaration, the name of the declared function may be used as `Context` to refer to that function. If several declarations match the same name, the `return@Context` is considered to be from the nearest matching function;
- If `return@Context` is used inside a non-labeled lambda literal, the name of the function **using** this lambda expression as its argument may be used as `Context` to refer to the lambda literal;
- If `return@Context` is used inside a labeled lambda literal, the label may be used as `Context` to refer to the lambda literal.

If a return expression is used in the context of a lambda literal which is *not inlined* in the current context and refers to any function scope declared outside this lambda literal, it is disallowed and should result in a compile-time error.

> Note: these rules mean a simple return expression inside a lambda expression returns **from the innermost function** in which this lambda expression is defined. They also mean such return expression is allowed only inside **inlined** lambda expressions.

8.26.3 Continue expressions

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`.

If a continue expression is used in the context of a lambda literal which refers to any loop scope outside this lambda literal, it is disallowed and should result in a compile-time error.

8.26.4 Break expressions

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`.

If a break expression is used in the context of a lambda literal which refers to any loop scope outside this lambda literal, it is disallowed and should result in a compile-time error.