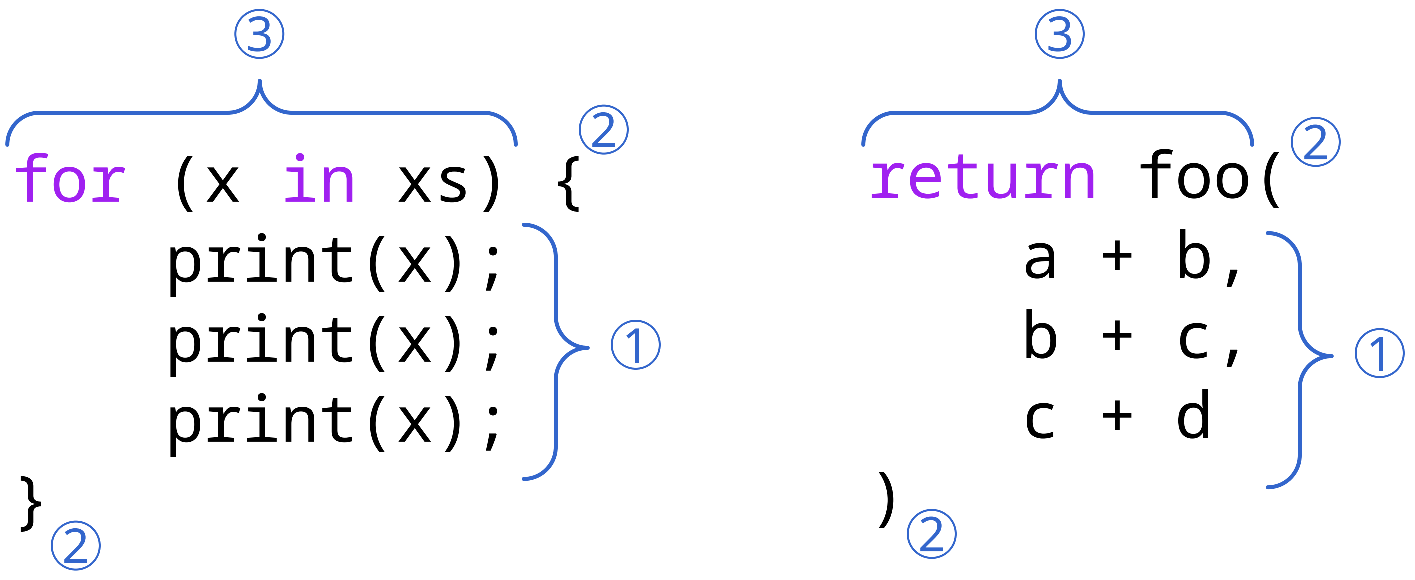
Indentation logic of `kotlin-mode`

2019-12-01

Basic idea (1/2)

Programs consist of...



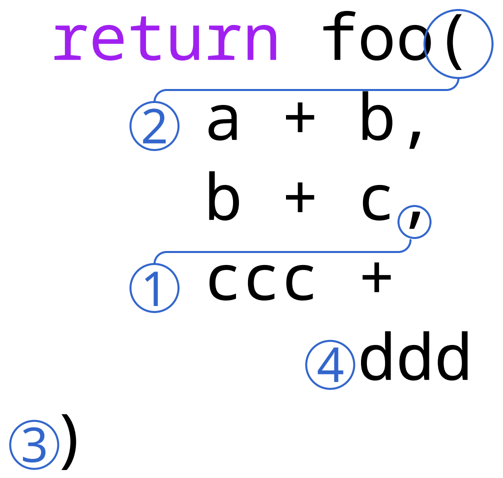
```
for (x in xs) {
    print(x);
    print(x);
    print(x);
}
```

```
return foo(
    a + b,
    b + c,
    c + d
)
```

List of statements/expressions ①,
sorrounded by (curly/round) brackets ②,
preceded by some texts ③

Basic idea (2/2)

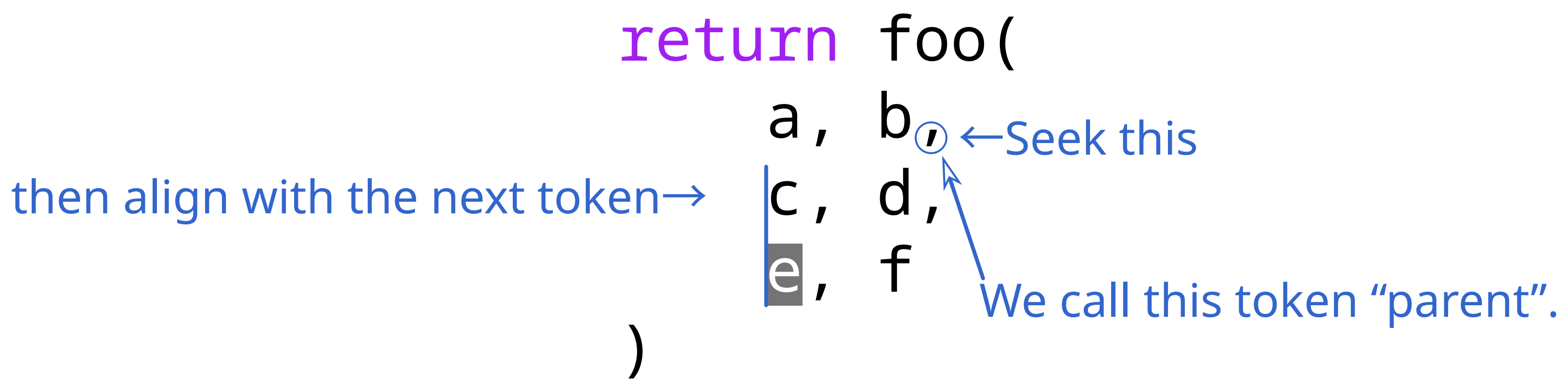
We have four cases to indent



```
return foo(
  ② a + b,
    b + c,
  ① ccc +
      ④ddd
③)
```

Case 1. after an element delimiter, such as semicolon or comma
Case 2. after an open bracket
Case 3. before close bracket
Case 4. other case, that is to say, inside a list element

Case 1: after an element delimiter



```
return foo(
    a, b,
    c, d,
    e, f
)
```

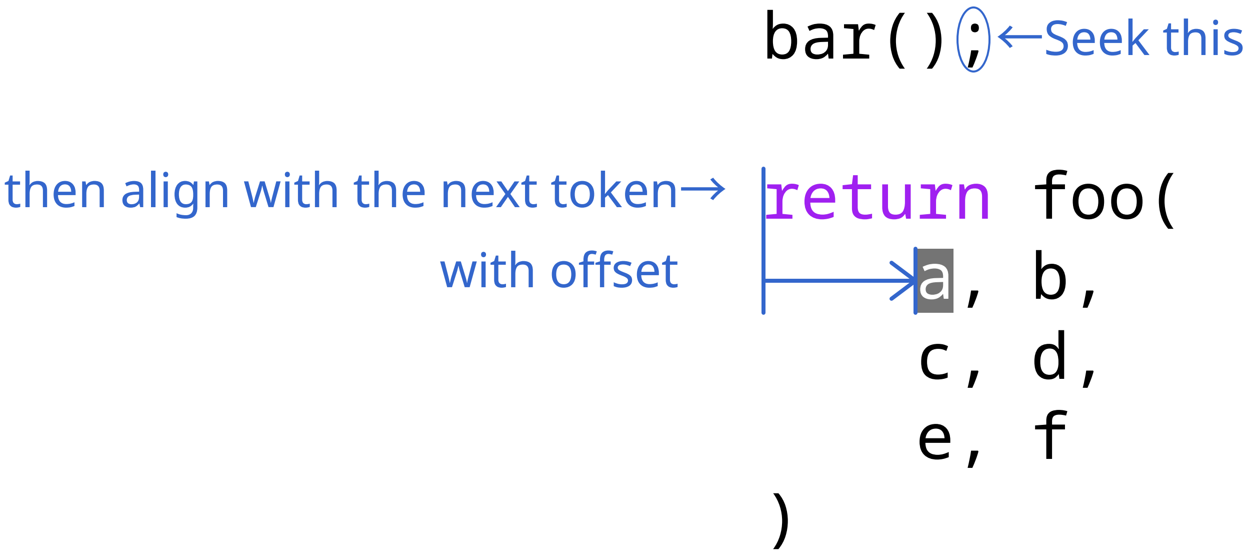
Align with the preceding element at the start of a line.

To seek the preceding element at the start of a line,
we seek an element delimiter (comma or semicolon)
at the end of a line or open bracket before the element.

Then the next token is the token we align to.

Elements may start with tokens of various types,
but it ends with tokens of handful types, so seeking it is easier.

Case 2. after an open bracket



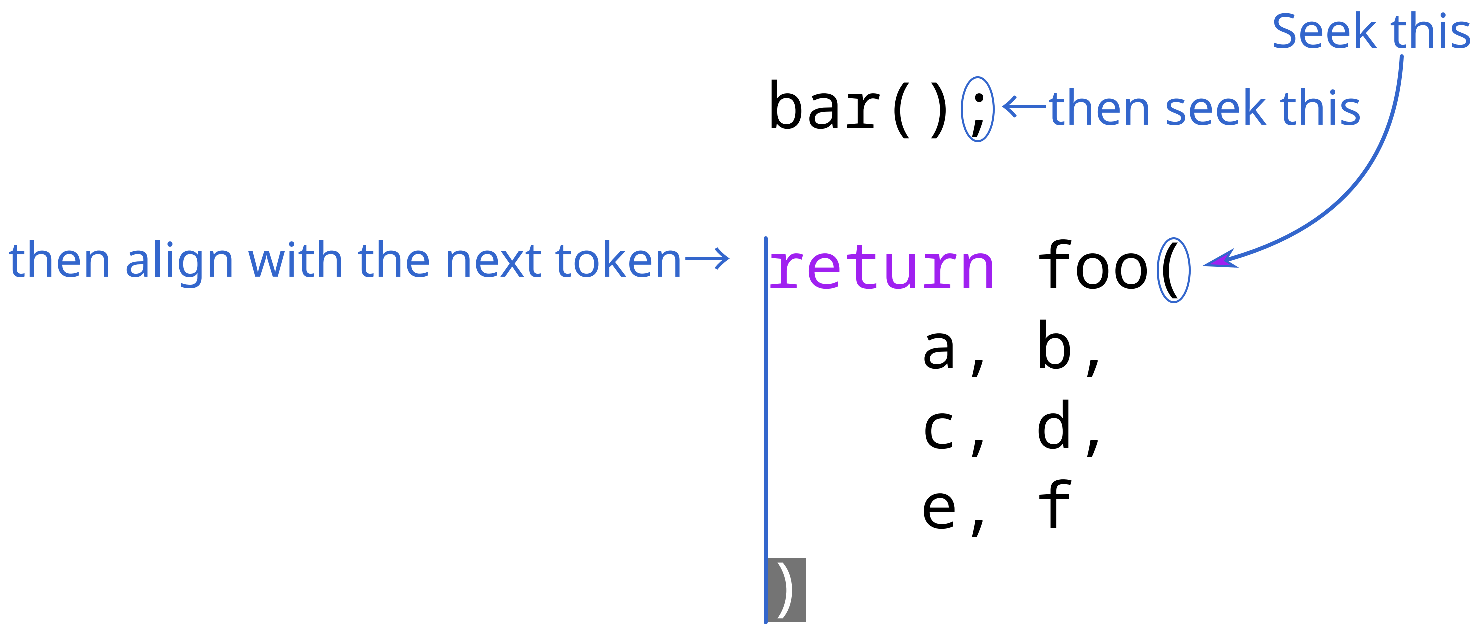
```
bar();

return foo(
         a, b,
         c, d,
         e, f
)
```

Align with the start of the “preceding text” with offset.

The procedure of seeking the start of the preceding text is same as the case 1. Seek a parent token, then the next token is the token align to.

Case 3. before close bracket



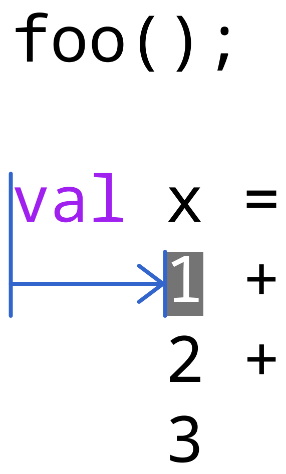
```
bar();

return foo(
    a, b,
    c, d,
    e, f
)
```

Align with the start of the preceding text of the open bracket without offset.

To find the open bracket, we can use the ‘`backward-list`’.

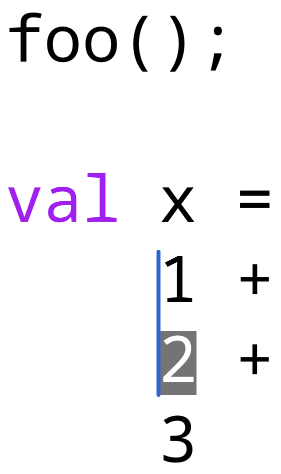
Case 4. other case, inside a list element



```
foo();

val x =
    1 +
    2 +
    3
```

If the line is the second line,
align with the start of
the element with offset.

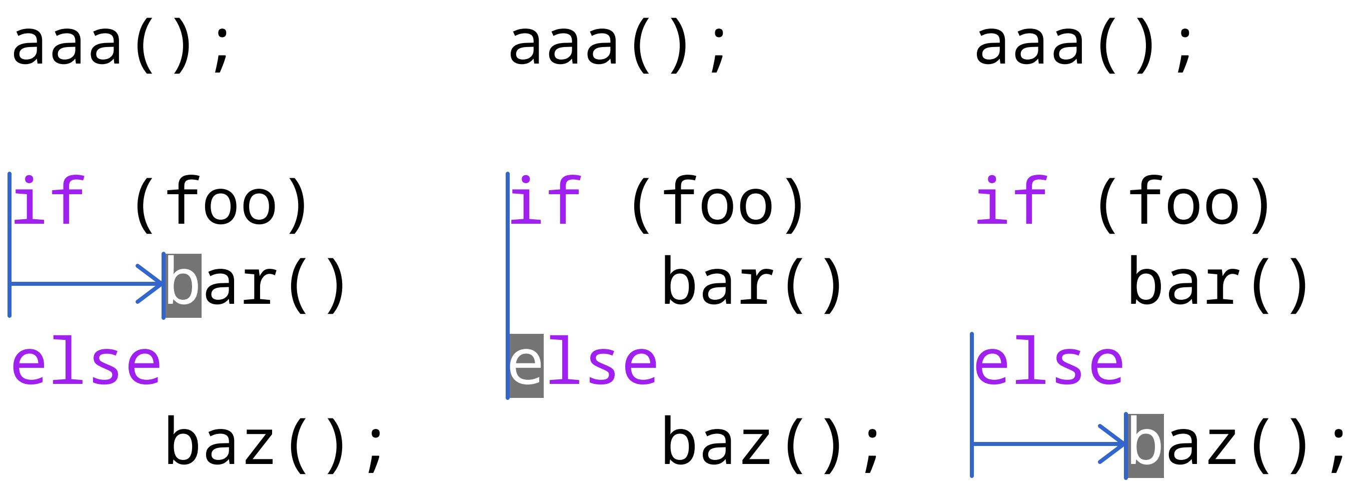


```
foo();

val x =
    1 +
    2 +
    3
```

If the line is the third or
following lines,
align with the the previous line.

if-else statement



```
aaa();

if (foo)
    bar()
else
    baz();
```

```
aaa();

if (foo)
    bar()
else
    baz();
```

```
aaa();

if (foo)
    bar()
else
    baz();
```

If the point is after `if (...)`,
then align with the `if` token with offset.

If the point is before `else` token,
then align with the matching `if` token without offset.

If the point is after `else` token,
then align with the `else` token with offset.

Note that `if`-`else` can be nested,
so when seeking the matching `if` token,
we have to count number of `else` and `if` tokens.

We have similar rules for `for`, `while`, and `do`-`while`.

Advanced topics

Implicit semicolons

In Kotlin and other languages, a statement may end with a newline.
We use a heuristic function ‘`kotlin-mode--implicit-semi-p`’
to detect it.
It examines tokens before and after the newline.

Example:

```
for (x in xs) {
    aaa()  ← Implicit semicolon here
    if (bbb)
        if (ccc)
            ddd()  ← No implicit semicolon here...
        else
            eee()  ← ... only here ...
    fff()          ← ... to aligh this line to the first if token
                     rather than the else token.
}  ← Implicit semicolon here
ccc()  ← Implicit semicolon here
```

Ambiguous commas, colons, curly brackets, and objects

Commas are not always contained by brackets.
Texts before brackets may contain another brackets.
We handle them carefully.

When seeking the previous element of this line,
if we got a pair of curly brackets,
then jump to the object token
and resume seeking,
to skip this comma.

```
class C: A by object: A1,
                      A2 {
             fun aaa() {}
         },
         B by object: B1,
                      B2 {
             fun bbb() {}
         } {
    fun ccc(x: X): Int {
        return when (x) {
            object: X1 by object: XX1 {
                        fun xxx1() {}
                    },
                    X2 {
                fun xxx2() {}
            },
            object: Y1,
                    Y2 {
                fun yyy() {}
            } ->
                1

            else ->
                2
        }
    }
}
```

Ambiguous arrows

Arrows have many meanings and indentation rules.
We use heuristics for this, but it is not precise.

```
val f = { g:
              (Int) ->  ← arrow for function type
              (Int) ->  ← arrow for function type
              Int ->    ← arrow for lambda parameters
    g(1, 2)
}

when (x) {
    1 -> ← arrow for when-entry
        f1 as (Int) ->  ← arrow for function type
              Int

    f2 as (Int) ->  ← arrow for function type
          Int ->    ← arrow for when-entry
        f3

    is (Int) ->  ← arrow for function type
       Int ->    ← arrow for when-entry
        f4
}
```

Cannot handle those cases for now.
We assume all arrows inside a when-expression are parts of when-entries.

Angle brackets <>

Token ‘<’ and ‘>’ may be used as inequality operators or angle brackets for type parameters.

We use heuristics to distinguish them:

- Angle bracket must be balanced.
- Angle bracket cannot contain some kind of tokens.

Ambiguous operators

We cannot handle those cases for now.

```
var shl = 1
val x = shl shl shl // The last “shl” is a variable named “shl”.
shl < 100 && foo() // This is not a continuation of the previous line.

var shl = 1
val x = shl shl // The last “shl” is a shift operator.
    shl < 100 && foo() // This is a continuation of the previous line.

var shl = 1
val x = shl shl shl ++ // postfix increment operator
shl < 100 && foo() // This is not a continuation of the previous line.

var shl = 1
val x = shl shl ++ // prefix increment operator
    shl < 100 && foo() // This is a continuation of the previous line.

val x = foo()!! // postfix operator
foo() // This is not a continuation of the previous line.

val x = !!  // two prefix operators
    foo() // This is a continuation of the previous line.
```

Implementation

Overview of functions for indentation. Details are omitted.

```
kotlin-mode--indent-line ← entry point for indenting line
  kotlin-mode--calculate-indent ← calculate the amount of the indentation
    kotlin-mode--calculate-indent-of-multiline-comment ← when the point is inside a multiline comment
    kotlin-mode--calculate-indent-of-multiline-string ← when the point is inside a multiline string
    kotlin-mode--calculate-indent-of-single-line-comment ← when the point is before a single-line comment
    kotlin-mode--calculate-indent-of-code ← other case, including before a single-line string
      kotlin-mode--forward-token ← lexer
        kotlin-mode--forward-token-simple ← lexer without unbounded recursion
        kotlin-mode--implicit-semi-p ← determinate implicit semicolon
        ...
      kotlin-mode--backward-token ← lexer
        kotlin-mode--backward-token-simple ← lexer without unbounded recursion
        kotlin-mode--implicit-semi-p
        ...
      kotlin-mode--calculate-indent-after-open-curly-brace ← when the point is after '{'
        kotlin-mode--curly-brace-type ← determinate the type of the block
        kotlin-mode--find-parent-and-align-with-next
          kotlin-mode--backward-sexps-until
            kotlin-mode--backward-token-or-list
              kotlin-mode--backward-token
            kotlin-mode--forward-token-or-list
              kotlin-mode--forward-token
      kotlin-mode--calculate-indent-after-comma
        ...
      kotlin-mode--calculate-indent-after-semicolon
        ...
      kotlin-mode--calculate-indent-of-expression
        ...
      kotlin-mode--find-parent-and-align-with-next
        ...
      ...
```

Data types

```
kotlin-mode--token
  Lexical tokens.  Consists of the type, the text, and the location (start and end) of the token.

kotlin-mode--indentation
  Location of anchor point paired with offset.
```

Other notable functions

```
kotlin-mode--indent-new-comment-line
  Replacement for indent-new-comment-line.  Break a line, indent it, and tweak comment delimiters.

kotlin-mode--post-self-insert
  Do electric indentation.
```