Literate Kotlin

Alexander Kuklev, JetBrains

We have a dream of making Kotlin a programming language suitable for all purposes in any context. Unfortunately, Kotlin in its current form is poorly suited for literate programming and lags far behind Python when it comes to illustrating ideas in tutorials and research papers. In this memo, we draft a Kotlin variant for literate programming and academic/educational use instead of ad hoc pseudocode.

When writing a computer science research paper or an educational tutorial, it's fine to spend days polishing code snippets for optimal readability, conciseness, and typographic perfection. Such applications value readability over writability, expressiveness over simplicity, principled considerations over practical concerns, and the avoidance of boilerplate and visual clutter at almost any cost. This seems to contradict one of the cornerstones of Kotlin: a remarkable balance between readability and writability, expressiveness and simplicity, orderliness and pragmatism, innovation and conservatism. But it turns out that the necessary changes, while fairly radical, are limited to syntax and default behavior. Literate Kotlin, the variant of Kotlin presented in this memo, can be seen as an alternative interface to the same underlying language.

Two first secions of the memo are devoted to syntax and appearance. The third section suggests some adjustments to the default behavior. In the last part, we discuss desirable semantic extensions that we believe will also benefit Kotlin itself in the long run.

1 Basic syntax and appearance

In 1984, Donald Knuth introduced literate programming, a practice of working not just on the source code but on a well-written and well-structured expository paper from which the source code can be extracted. The ultimate result should be the expository paper, which carefully walks through all the nooks and crannies of the source code, explaining the ideas, and documenting the reasoning behind certain decisions. It is both at the same time: an essay interspersed with code snippets and a source code interleaved by accompanying text.

Existing programming languages treat the accompanying text as a second-class citizen, as 'comments' bashfully fenced with freakish digraphs like `/* … */`. Markup languages used for writing computer science research papers (mainly (La)TeX) and tutorials (mainly HTML and Markdown) take the opposite approach, treating code snippets as second-class citizens. We propose a balanced approach treating code and text on a par. Before we can present it, we need to explain our treatment of blocks and literals.

1.1 Blocks

We propose restricting the use of braces only for inline blocks and using the off-side rule for multiline blocks. The indentation-based structure sticks out above everything else, so it should take precedence over comments, quoted literals, and brackets. **This massively speeds up incremental parsing: blocks can be recognized instantly without prior parsing and processed independently.**

We propose to fix the block indentation to two whitespaces once and for all, any other indent (1 or >2) continues the previous line:

```
fun example(files : List<File>,
            target : File)
  ...
  return something + somethingElse + somethingOther +
   yetSomething + rest
```

IDEs should provide visual reading aid for consequent dedents by displaying end marks (■).

```
fun main(args : List<String>)
  for (arg in args)
    println(arg)
  ■
```

At the end of large indentation regions, labeled end marks (e.g. `■ main`) should be used.

1.2 Unquoted literals

In Kotlin, trailing functional arguments enjoy special syntax: `a.map({ println(it) })` is simply `a.map { println(it) }`. Trailing `String` arguments (in general, `AdditionalContext.()→String<INTERPOLATION_STYLE>`) deserve special syntax too. Unquoted literals begin with a left-flanking `~` followed by a whitespace or a line break. They end just before the next line with an indentation level less or equal to that of the line the literal starts. Line breaks can be `\`-escaped, `\{…}`-syntax used for type-based (e.g. `String<SQL>`) JSR 430-like safe interpolation.

```
fun greet(name : String)
  println~ Hello, \{name}!
```

Those would also work nicely with key-value lists:

```
address: Address
  house:~
    Olaf Taanensen
    Tordenskiolds 24
  city:~ Oslo
```

1.3 Comments

Our proposal from the first section implies mandatory indentation for all non-inline blocks. Thus, all remaining unindented lines are top-level definitions (`class …`, `object …`, …) and directives (`package …`, `import …`). These necessarily begin with an annotation or a keyword. Annotations readily begin with an `@`, and it won't be too much pain to prepend `@` to top-level keywords: `@import` already looks familiar from CSS, `@data class` and `@sealed class` make perfect sense anyway: most modifier keywords are nothing but inbuilt annotations.

In this way, every code line either starts with an `@`, or is an indented line following a code line (with possibly one or more blank lines in between). Let us require the compiler to skim all the lines that do not meet this specification. These other lines can now be used for the accompanying text written "as is" without fencing. We suggest using (La)TeX hybrid-mode Markdown (`\usepackage[hybrid]{markdown}`): it has excellent readability while providing the whole power of (La)TeX, the de facto standard for writing technical and scientific papers.

Freely interleaving the code and accompanying text, without fencing either, is the perfect fit for literate programming. The very same file can be either fed into a Kotlin compiler to produce a binary or into a Markdown/TeX processor to produce a paper.

Sometimes, it is still desirable to comment on a single line. Since at least 1958, em-dashes – surrounded by whitespaces have been used for single-line comments to separate code and text. It seems to be a typographically perfect solution, but the standard PC keyboard layout lacks em-dash. Ada, Agda, SQL, Eiffel, Elm, Haskell, Lua, SQL, and several other languages use double dash `--` as an ASCII substitute for em-dashes, but this is incompatible with the C-style decrement operator. We propose to use either the real em-dash (which is present in the standard MacOS keyboard layout and can be accessed via Compose+`- - -` on Linux) or a single backtick surrounded by whitespaces.

1.4 Plain text notebooks

Jypiter-style notebooks can be seen as an interactive form of literate programming. The expository paper can (and should) contain runnable code samples to illustrate usages of the code being explained and test cases for each non-trivial function. These should be optimally displayed as runnable, editable, debbugable blocks with rich (visual, animated, interactive) output, that's what notebooks are build from. Since we see such blocks as an element of literate programming, we want to provide plain text syntax for them:

```
@run sampleFunction(1, 3)

@run 1 + 2 + 3
@expect 6

@run `Named sample`:
  val a = 1 + 2
  a + 3

@run(collapsed: true, autoexec: false)
  someLenghtyComputation()
```

2 Syntaxtic and typographic sugar

2.1 Pipeline notation

In mathematics and functional programming, it's fairly common to use the right pointing black triangle for inverse application, i.e. `x ▸ foo ▸ bar ≔ bar(foo(x))`, which gives an intuitive processing pipeline syntax. We propose to display `x.let f` as `x ▸ f` and `x?.let f` as `x ▸? f`, with mandatory whitespaces to disambiguate from the syntax we propose in the next paragraph.

In most cases, processing pipelines in Kotlin are also incluse method invocations. In Kotlin, `obj.foo(…)` can mean both invocation of the method `foo` and application of the property `foo` of a callable type. Displaying dots as ▸ in case of methods follows the long tradition of using arrows for child methods started by PL/I in the late 60s. It helps disambiguating between properties and methods, and leads to typographically perfect pipeline syntax:

```
files ▸dropLast(n) ▸withIndex ▸last
```

```
fun example(files : List<File>,
            target : File)
  files ▸filter
    it.size > 0 &&
    it.type = "image/png"
  ▸map { it.name }
  ▸withIndex ▸map fun(idx, item)
    ...
  ■
```

NB. Moving the safe call question mark to the right (cf. `as?`-operator) allows displaying `…OrNull` methods as `…?`, e.g. `▸first?` instead of `.firstOrNull`, `a[i]?` instead of `a.getOrNull(i)`, etc.

2.2 Ad hoc infix operators

Pipeline notation provides an aesthetically pleasing way to act on one object, but sometimes several objects have to be fused, which is best expressed by infix operators. We propose turning any binary (or vararg) function into an infix operator with chevrons (not <angular brackets>!):

```
a ‹and› b          2 ‹Nat.plus› 3          users ‹join(::id)› customers
```

2.3 Reducing type annotations

Many functional languages allow one to declare multiple consecutive variables of the same type separating them by whitespaces

```
fun plus(x y : Int) : Int
```

and declaring default types of variables based on their names:

```
reserve z : Point, prefix n : Int, suffix count : Int
```

After this declaration, identifier `z` with optional numeric indices (e.g. `z2`) will have default type `Point`, and all multipart identifiers starting with first part `n` or last part `count` (e.g. `nUsers` and `pointCount`, but not `neighbour` or `account`) will have default type `Int`. Generalized form of reserve blocks may greatly simplify signatures of generic methods, see https://agda.readthedocs.io/en/v2.7.0/language/generalization-of-declared-variables.html.

2.4 Compliance with mathematical notation

To improve readability, reduce ambiguities, and comply with established mathematical notation, we require mandatory whitespaces around all infix operators and relations including `n : Int`, but excluding `a·b`, `a..b`, and `a..<b`.

Multiplication should be displayed as `·`, comparison operators as `≤`, `≥`, `=`, `≠`, logical operators as `¬`, `∧`, `∨`, arrow in function literals as `{ x ↦ x + 1 }`, the assignment operator as `≔` when introducing a fresh name (e.g. `val a ≔ 5`), and by left-flanking colon `key: value` otherwise. Additionally, we propose two opt-in features:

- `import CoefficientNotation` (used in algebra) to interpret `2x` for `2·x`
- `import SegmentsNotation` (used in geometry) to interpret runs of uppercase letters, possibly with indices, (`ABC`, `ABCD`, `X1X2`) as `Segments(A, B, C)`, `Segments(A, B, C, D)`, `Segments(X1, X2)`. Uppercase identifiers are still available with backticks (`` `ABC` ``).

In Kotlin, operators are always referred to by their verbatim name, like `minus`, `unaryMinus`, and `dec`. In mathematics, it is customary to allow symbolic references. We propose operator symbols enclosed in parentheses with no whitespaces around for infix operators, whitespace before for postfix ones, and whitespace after for prefix ones: `::(-)` = `::minus`, `::(- )` = `::unaryMinus`, and `::( --)` = `::decrement`.

2.5 Compliance with functional notation

In Kotlin, the method invocation `method(args)` is a complex syntactic entity, supporting optional arguments, named arguments, and variable number of tail arguments, as well as special handling for the last argument of functional type. Parentheses can be omitted (while invocation still is implied!). For that reason, methods be referred to by their name, and the notation `::method` (`class::method` in fully qualified case) has to be used instead.

Application of callables (values of type `(args)→ R`) mimics method invocation with the exception that parentheses are mandatory and several subtle limitations. This approach contradicts the usual mathematical practice, where it is customary to write `sin x` instead of `sin(x)` and `f a b` for `( f(a) )(b)`. We propose to use opt-in `import FunctionalNotation` adding the type former `X → Y` (without parens around `X`) to introduce functions like `sin` that can be used as customary in mathematics and functional programming languages.

2.6 Dual naming: verbose names and concise names

Naming things is hard both in programming and in mathematics. Objects and operations should have readable and self-explanatory names. However, verbose names may severely impair readability in formulas. Compare the following three variants of the same formula:

- `n·(n + 1) / 2`,
- `elementCount * (elementCount + 1) / 2`, and
- `div(times(elementCount, plus(elementCount, 1)), 2)`

Dual naming `` `verbose name`conciseName `` is a way to reconcile contradictory requirements.

```
val `element count`n ≔ ...
val (`height`x, `width`y) ≔ o.getDimensions()
class List<`element type`T>
```

2.7 Unicode abbrevations and custom operators

It should be allowed to use non-ASCII characters and custom operators as `conciseNames`. Readable `verbose name` is strictly necessary (so one knows how to read those symbols aloud) and ASCII-only if `conciseName` contains characters not available on a standard keyboard.

```
enum class `Boolean`𝔹 {`true`, `false`}

data class `Pair`(×)<out X, out Y>(val first : X, val second : Y)

val `factorial`( !) ≔ fun(n : ℕ)
  when(n) { 0 ↦ 1; p⁺ ↦ n · p! }

val `conjugate`(+ ) ≔ fun(c : ℂ)
  Complex(c.re, -c.im)
```

Now we can use 𝔹 for `Boolean`, `X × Y` for `Pair<X, Y>`, `n!` for `factorial(n)`, `+c` for `conjugate(c)`.

The verbose name can be a “closed operator”:

```
fun <T> `if $c then $a else $b`ifelse(a b : T, c : 𝔹) : T

fun `⌊$x⌋`floor(x : Float)
```

2.7.1 Operator tightness

Expressions like `+n!` can be parsed both as `( +n )!` and `+( n! )`. With definitions as above, it is not a valid expression, it’s a `syntax error: ambiguous expression`. However, one can specify the tightness for the operators. If `( !)` binds tighter than `(+ )`, `+n!` resolves into `+(n!)` and the other way around.

Infix operators may have different right and left tightness. For example, `(-)` binds tighter on the right than on the left: `a - b - c` resolves into `(a - b) - c`.

To specify tightness, we allow introducing abstract tightness levels called Operator Categories and declaring them to be tighter or weaker than some other levels. They must merely form a directed acyclic graph and do not have to be pairwise comparable.

Actually, an `OperatorCategory` is a bit more than a label: it specifies how to deal with respective homogeneous operator chains. For example, there is a large operator category `EqRel` that contains comparison operators and resolves their chains `a < b < c` into `(a < b ‹and› b < c)`.

2.7.2 Operators with inner parameters

Operators may have inner parameters, e.g. the indexed access operator `arr[i]` is a postfix operator with an inner parameter `( [$idx])`. In mathematics, many binary operators, including tensor product and semidirect product, have optional parameters rendered as subscripts or superscripts.

Using parser techniques developed for the Agda programming language, we can embrace this complexity without any considerable problems.

By combining custom `OperatorCategory` and operators with inner parameters, one can even embrace the notorious example of insane operator complexity: the METAPOST path notation:

```
draw a -- b -- c --cycle              — A triangle, (--)-lines are straight
draw a ~~ b ~~ c ~~cycle              — A circle through abc, (~~)-lines are curved
draw a ~~ b ~~ c ~- d -- e --cycle    — (~-) connect smoothly only on the left side

draw a ~~ b ~~[tension: 1.5, 1]~~ c ~~ d
draw a [curl: k]~~ c ~~[curl: k] d
draw a ~~ b [up]~~ c [left]~~ d ~~ e.
draw (0,0) ~~[controls: (26.8,-1.8), (51.4,14.6)]~~
 (60,40) ~~[controls: (67.1,61.0), (59.8,84.6)]~~ (30,50)
```

2.8 Let blocks

We suggest introducing let-blocks. Let-block header contains a list of vals being defined, the following block contains a list of conditions those have to satisfy.

```
let x y : Float
  x + 2y = 5
  x - y = 4
```

A let-block compiles if there is a compiler solver-plugin that supports given condition forms and succeeds iff there is a unique or a preferred solution.

We envision at least two solvers: Linear solver precisely as in Knuth's METAPOST (in particular, solves the example above) and, in the distant future, a deep unification solver as defined in The Verse Calculus paper[1] by Simon Peyton Jones, Guy Steele et al., that possesses enormous expressive power, elegantly subsuming both Prolog and Datalog.

3 Changing default behavior

While being very radical, all of the above suggestions are merely syntactic surgar, they are almost exclusively limited to parser and IDE. Yet they are not quite enough to make Kotlin appealing as a substution for "pseudocode".

3.1 Pythonic integers

Academic pseudocode assumes the default integer type to be overflow-free as in Python. The operator `(/)` is always used as the true division operator even when both operands are integer. For integer division, an additional operator `(//)` (as in Python) should be introduced.

3.2 Operator attribution and type classes

In accordance with their mathematical semantics, expressions like `2 + 3` should be interpreted as `Int.plus(2, 3)` rather than `2.plus(3)`, i.e. arithmetic operators are considered to be properties belonging to companion objects of the given numeric type rather than methods of number objects themselves.

[1] https://simon.peytonjones.org/assets/pdfs/verse-icfp23.pdf

4 Semantic extensions

4.1 Type classes

Since we mentioned companion objects containing operators like "plus", we should also mention that the notion of type-classes is indispensable in many academic contexts. In Kotlin, one can define both nested classes and extension functions. The type-classes are, in a sense, extension nested data classes with quite a bit of additional syntactic sugar.

Consider the following definition of a monoid structure on a type `T`:

```
data class <T>.Monoid(val compose : (vararg xs : T)-> T)
  val unit ≔ compose()     — Unit is the nullary composition

  contracts {
    unit ‹compose› x = x
    x ‹compose› unit = x
    x ‹compose› y ‹compose› z = x ‹compose› (y ‹compose› z)
    compose(x, *xs) = x ‹compose› compose(*xs)
  }
```

With such a definition, we now can write functions like this:

```
fun <T : Monoid> square(x : T)
  x ‹T.compose› x
```

or, equivalently

```
fun <T : Monoid<::(∘)>> square(x : T)
  x ∘ x
```

— here, generics resolution implicitly put both `T` and `(∘)` into the scope.

Support for higher kinds and type class inheritance can be directly modeled after Arend.

4.2 Dependent types

Eventually, one should carefully introduce full-blown dependent types, following the defensive approach to dependent types pioneered in Haskell.

Amusingly, adding dependent types to Kotlin immediately allows embedding SQL-type queries almost verbatim:

```
fun Table.select(cols : this.colsCtx.()-> List<t.Col>) : LazyTable
fun LazyTable.where(clause : this.ctx.()-> Boolean) : LazyTable

users ▸select { name, age, address as "userAddress" }
      ▸where { age > 18 }
```

4.3 Runtime-introspectable coroutines

We suggest using labeled blocks (e.g. `name@ { code }` in coroutines as runtime-introspectable execution states. If the coroutine job `j` is currently running inside of the labled block `EstablishingConnection@`, we want `(j.state is EstablishingConnection)` to be true. The hierarchy of nested blocks in the coroutine should autogenerate a corresponding interface hierarchy.

Furthermore, propose coroutines to have exposable read-only data-only properties that can be used to track the coroutine progress. We suggest allowing visibility modifiers `public` and `internal` for top-level `var`s and `val`s as well as the ones in labeled blocks and labeled loops:

```
val j ≔ launch
  ...prepare data
  Moving@ for (i in files.indices)
    public val progress = i / files.size
    fs▸move(...)
  ...finalize

val u ≔ launch
  ...
  when (val s ≔ j.state)
    Moving ↦ println~ Moving files, \{s.progress · 100}...
```

To avoid misalignment, `j.state` must read out the properties immediately when invoked; all properties must be data-only, i.e. either of primitive type, or hereditarily immutable.

4.4 Strong object typing

Eventually, structured concurrency should be generalized to structured ownership, with a general notion of managed object and managing scopes. Kotlinesque coroutine scopes and Rustacean lifetimes are managing scopes, jobs and shared mutable variables are respective managed objects, governed by separation logic. Redistributable references to managed objects can be faithfully treated as values, types of which are path-dependent (in Scala sense) on their respective managing scopes (cs.Job, lt.Var). Thus, to handle them, it would suffice to support full-blown PDTs and allow passing objects (coroutine scopes, lifetimes, etc.) not only as arguments, but alternatively as parameters, e.g. `fun <cs : CoroutineScope> example(v : cs.MutRef<Int>)`.

Besides managed objects, there are exclusively owned objects (cf. uniqueness typing). References to those cannot be copied or passed arbitrarily, so they must be marked syntactically as being non-values. When a method gets them as arguments, the respective arguments must be annotated either `my obj` or `borrow obj` in case the object is returned back to the call site after completion. A local "variable" containing an exclusively-owned object should be declared `my obj` instead of `val obj`, e.g. `my job = lunch someCoroutine(…)` or `my o = object : SomeInterface {…}`. Exclusively owned objects appear most frequently as receivers (`this`). Owing to smart casts, strong typing for exclusively-owned objects can be piggybacked on the existing Kotlin type system by extending the syntax and semantics for interfaces. The resulting type system fragment would closely reassemble the system by F. Pfennig and A. Das from "Verified Linear Session-Typed Concurrent Programming[2]", see also https://www.cs.cmu.edu/~fp/papers/lmcs22a.pdf for a primer on possible concise syntax.

The third kind of objects are the external/standalone objects (resources), such as filesystem and database: those are properly handled by a capability system like that in Scala 3.

5 Conclusion and outlook

In this memo, we have outlined the vision and rationale behind Literate Kotlin, a variant of Kotlin tailored for literate programming and academic use. By addressing the limitations of Kotlin in its current form, we aim to bridge the gap between the language's inherent strengths and the specific needs of educational and research contexts.

Our proposed changes, while radical, are superficial and in the most part easy to implement. We believe that by enhancing readability, expressiveness, and typographic quality according to our propositions, Literate Kotlin can serve as a powerful tool for educators, researchers, and anyone who values clarity and precision in code presentation.

[2] https://www.cs.cmu.edu/~fp/papers/ppdp20.pdf

The adjustments to syntax and appearance, along with the suggested behavioral modifications and semantic extensions, are designed to make Literate Kotlin a viable alternative for those who currently rely on pseudocode or other languages for illustrative purposes. We are confident that these enhancements will not only benefit the academic community, but also contribute to the broader Kotlin ecosystem by promoting a more versatile and expressive language.

The early drafts of this memo were enthusiastically received at the Department of Software Science at Radboud University, the Department of Informatics of the Göttingen University, and the Department of Mathematics at TU Dresden. As we move forward, we invite the academic community to engage with Literate Kotlin, provide feedback, and contribute to its evolution. Together, we can realize the dream of making Kotlin a truly universal programming language.