Literate Kotlin

Alexander Kuklev[1,2] ‹a@kuklev.com›

[1]Radboud University Nijmegen, Software Science
[2]JetBrains Research

Kotlin in its current form is not fully suited for literate programming and lags 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 as 'runnable pseudocode'.

These applications are very peculiar as they favor readability of carefully written code above everything else. When writing a research paper or an educational tutorial, it is quite common to spend days polishing code snippets for optimal readability, conciseness, and typographic perfection. We propose a series of adjustments mostly limited to the syntax and default behavior to allow for perfectly polished code and seamless interleaving of code and text. Literate Kotlin, as we currently call it, can be seen as an alternative front-end to the same underlying language.

1 Appearance

Sticking to typographic standards in scientific publishing and adopting confusion-reducing approaches from Python could make Kotlin more approachable for a wider audience.

We suggest the following pretty-printing: `2a·b` for multiplication, `{ x ↦ f(x) }` for closures, `≤`, `≥`, `≠`, `=` for comparison operators, `set x = 2x + 1` and `set obj.counter += i` for assignment, `def` for `fun` in definitions, `Int16/32/64` for `Short/Int/Long`, and `//` for integer division, while `Int` and `/` are reserved for "true" integers and division as in Python, and line comments use `#` `2+` whitespaces apart from the code. Except for `x·y`, `p/q`, and range operators `a..b`, and `a..<b`, we propose mandatory spaces around infix operators and relations including `n : Int`. Trailing colons (`name: value`) should be used for named arguments/fields. We also suggest introducing support for positional-only and name-only arguments and deprecating ASCII logical operators.

1.1 Blocks

To improve readability and reduce visual clutter, we propose using the off-side rule for multiline blocks, while limiting the use of braces for inline blocks only and deprecating `/* */`-comments and `"""`-literals in favor of "tidy" literals and intertext (see next sections). To 'comment out' parts of code, we suggest nestable deletion braces `{- -}` that must be either used inline or placed on lines of matching indentation. The indentation-based structure is the most noticeable part of formatting, so it should also have precedence over parentheses. Prioritizing indentation over bracketing has a nice side effect of substantially speeding up incremental parsing by IDEs.

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

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

Pretty-printing should provide visual reading aid for consequent dedents by displaying end marks (■) commonly used in pseudocode and mathematics alike:

```
def 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 Tidy trailing, multi-line, and keyword literals

Kotlin already has a simplified syntax for trailing functional arguments: `a.map { println(it) }` stands for `a.map({ println(it) })`. We propose a similar syntax for trailing `String` arguments (`AdditionalContext.()→ String<INTERPOLATION_STYLE>` in general), covering both rest-of-line and multiline literals. Trailing `~` followed by a space or a line break opens a literal stretching to the nearest line with an indentation level not exceeding that of the line the literal starts. Line breaks can be `\`-escaped. We propose `\{…}`-syntax for JSR 430-like type-based (e.g. `String<SQL>`) safe interpolation and formated interpolation, while deprecating unsafe raw `$`-interpolation.

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

This approach also works nicely with property lists, yielding a "better YAML":

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

Apostrophes can be used without ambiguity for both character literals `'x'` and keyword literals when placed in front of an alpha-numeric string, e.g. `('usd, 'eur)`, `lang: 'en-US`.

1.3 Functional notation

We propose adding the type former `X → Y` (in addition to extant `(Xs)→ Y`) for functions supporting functional style application, e.g. `sin x` for `sin(x)` and `f a b` for `( f(a) )(b)`. To get the best of both worlds, we propose allowing such functions and infix keyword operators to additionally have optional "subscript parameters" written as `log_(10) x`, that takes full advantage of the usual method invocation syntax with both positional and named arguments.

1.4 Pipeline notation

In mathematics and functional programming, it's fairly common to use the right-pointing black triangle for inverse application, allowing an intuitive pipeline notation: `x ▸ foo ▸ bar` means `bar(foo(x))`. We thus suggest pretty-printing `(x.let f)` as `x ▸ f` and `(x?.let f)` as `x ▸? f`.

In contrast to purely functional languages, pipelines in Kotlin primarily consist of method invocations. In Kotlin, `obj.foo(…)` can mean both invocation of the method `foo` and application of the property `foo` of a callable type. Following the long tradition started in the late 60s by PL/I, we propose to display dots as `▸` when invoking methods, in this case with no trailing whitespace. It solves invocation/application-ambiguity and gives typographically perfect pipeline syntax:

```
files ▸filter
  it.size > 0 &&
  it.type = "image/png"
▸map { it.name }
▸withIndex ▸fold(0) fun(acc, item)
  ...
▸first?
```

With `▸?` for `?.`, we can pretty-print `.firstOrNull` as `▸first?`, `a.getOrNull(i)` as `a[i]?`, etc.

A combination of above features (together with type providers described elsewhere) enables an ergonomic and type-safe `SQL`-like integrated query notation:

```
(users join_{id} accounts)
▸select { users.name, accounts.name as 'login, age, address }
▸where { login ≠ null and age > 18 }
```

2 Literate programming

We do not only aim at making Kotlin suitable for writing perfectly looking and perfectly readable code snippets, but also for literate programming, the only known technique to produce genuinely maintainable and thoroughly auditable software. Literate programming — introduced by Donald Knuth in 1984 — is 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 subtleties of the source code, explaining the ideas, and documenting the reasoning behind certain decisions. It is, at the same time, both an essay interspersed with code snippets and a source code interleaved by accompanying text.

Mainstream 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 (e.g. TeX and LaTeX) and tutorials (e.g. HTML and Markdown) take the opposite approach, treating code snippets as second-class citizens. We propose a balanced approach to treating code and prose on a par.

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, as 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 a flavour of LaTeX-hybrid Markdown (`\usepackage[smartHybrid]{markdown}`): it has excellent readability while providing all the power of LaTeX, the golden standard for writing technical and scientific papers.

Freely interleaving the code and accompanying text, without fencing, 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 an expository paper.

2.1 Kotlin-flavoured markdown

The default hybrid mode of the TeX `\usepackage{markdown}` package is not entirely satisfactory for our purposes, so we developed replacements for several of its options: `smartHybrid` (refines `hybrid`), `fencedEnvs` (refines `fencedDivs`), and `offsideCode` (refines `fencedCode`).

`offsideCode` recognizes indented blocks starting with `@keyword` (e.g. `@class List<T>`) as code blocks. That's how we implement document generation for Literate Kotlin sources! Just typeset them with a preamble containing `\usepackage[offsideCode,…]{markdown}`.

`fencedEnvs` is an improvement of the `fencedDivs` option that works as follows:

```
::: boxed           ⟩     \begin{boxed}           | <boxed>
Some _text_.        ⟩       Some \emph{text}.     |   Some <em>text</em>.
:::                 ⟩     \end{boxed}             | </boxed>
```

For envs like `theorem` and `table` that come in both numbered and unnumbered variants, titlecase (e.g. `Figure`) is used for the numbered and lowercase (e.g `figure`) for the unnumbered one.

`smartHybrid` is an improvement of the `hybrid` option: just like `hybrid`, it allows TeX commands in Markdown but uses the percent sign (e.g. `%newpage`) as sigil instead of `\` backslashes to avoid collisions with `\`-escaping in Markdown. Indented blocks starting with `%EnvName` are translated into LaTeX environments with their content preserved verbatim.

```
::: Figure[hb] Sample Caption            ) \begin{figure}[hb]\caption{Sample Caption}
%tikzpicture                             ) \begin{tikzpicture}
  \draw[gray, thick] (-1,2) -- (2,-4);   )   \draw[gray, thick] (-1,2) -- (2,-4);
  \draw[gray, thick] (-1,-1) -- (2,2);   )   \draw[gray, thick] (-1,-1) -- (2,2);
  \filldraw[black] (0,0) circle (2pt);   )   \filldraw[black] (0,0) circle (2pt);
:::                                      ) \end{tikzpicture}\end{figure}
```

The usage of percent signs does not cause problems, as they are used in TeX only for comments.

We have not yet implemented a Literate Kotlin → HTML processor, but we intend to translate LaTeX-commands and environments into HTML tags `<figure parameters="hb"><caption>Sample caption</caption><tikzpicture data="..."/></figure>` that can be then rendered by any of the respective frameworks like Vue.js, Riot.js, etc. It would not be possible to match TeX in typographical perfection because no web browser engine supports or even plans to support proper hyphenation, microtypography, tabbing, etc. in the foreseeable future. However, HTML has a different strength: the potential for interactivity. Eventually, we hope to develop a documentation generator that turns Literate Kotlin into interactive online documentation with interactive code snippets, similar to the Kotlin Tour.[1]

2.2 Plain text notebook format

Jupyter-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, debuggable blocks with rich (visual, animated, interactive) output. That's precisely what the so-called cells in Jypyter-like notebooks are. Since we see such cells 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()
```

3 Conclusion and outlook

We have outlined the vision and rationale behind Literate Kotlin, a variant of Kotlin tailored for literate programming, academic, and educational 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.

This memo is the first in a series dedicated to Literate Kotlin. The proposals presented here have been deliberately selected to maintain full bi-convertibility with the original Kotlin, and we believe they are sufficient to make it best suited for literate programming and a viable alternative for those who currently rely on pseudocode or other languages for illustrative purposes. We are confident that these efforts will not only benefit the academic community but also contribute to the broader Kotlin ecosystem by promoting a more versatile and expressive language.

[1] https://kotlinlang.org/docs/kotlin-tour-hello-world.html