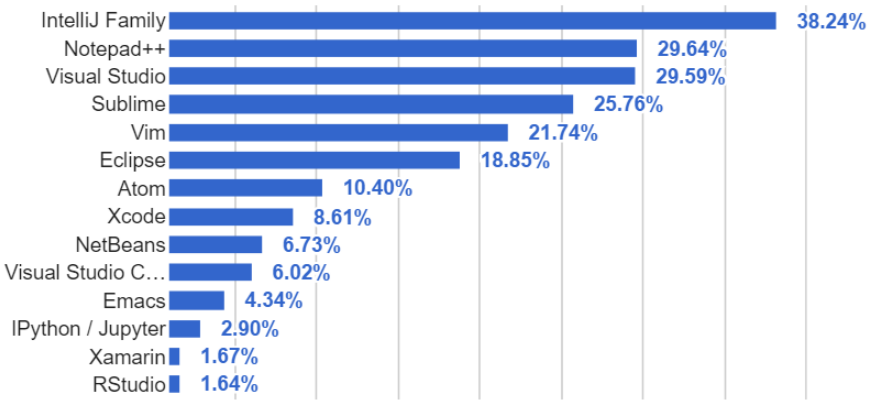
Code is not just text

Or why our code editors are inadequate tools

Let's look at some numbers:



Usage of Development Environments (Source: I plotted it with data from the Stack Overflow Developer Survey 2016. Keep in mind that people use many tools, hence the numbers add up to more than 100%)

What do all of these have in common? They're all text editors. "But wait, MyIDE can do many more things!", you say. And you're right. IntelliJ sure is awesome. But even when you're programming in your favorite IDE, you're still just manipulating text, despite all the fancy navigational and code completion features. So what's wrong with that?

The problems with editing code like text

> *Do we want programmers in 2050 to still have to deal with missing semicolons?*

In programming, we all start as noobs fighting with **Syntax Errors**. The first programs people interact with tend to be Browsers, Search Engines, Messengers, Social Networks and not Code Editors. What makes the latter so very different is that while they also appear to accept free-text, they actually have hard syntactic rules about what input is accepted. If you're lucky, you'll see the resulting errors in-line, but sometimes even the best parsers can't recover.

After surviving the first dozens of hours of syntactial failing, the gramatical rules of that language become ingrained in the programmer's mind and there are fewer and fewer syntax errors. They never fully disappear though, not even pros are immune to typos/slips and it doesn't require a LISP to get your braces wrong. Every time you use more than 4 consecutive braces you have to manually count,

instead of quickly getting an accurate unconcious count from your brain (known as Subitizing).
It might “only” be a daily nuisance for professional programmers, but for the novice having to learn an arcane syntax while also getting comfortable with computational thinking, might just be a fatal barrier.

And that’s just the start of the daily absurdity of a programmer’s life: Tabs vs. Spaces? Where to put my opening/closing braces? How do I order my imports? What should be the character limit per line? Do I space around my control statements? Should I use trailing commas? …
Even if you don’t take part in these bikeshedding discussions, as a responsible coder you still have the extra mental load to take care to stay consistent with the project’s code style.
Now I’m not saying that there aren’t solutions to these problems. There is EditorConfig for consistent indentation. In the JavaScript world we have ESLint to enforce all kind of code rules and based on it StandardJS emerged hoping to create a new code style standard. Both offer command line options to not only nag you about your inconsistent code but actually try to fix it. Just a few weeks ago Prettier appeared which fixes by default and also takes care of breaking your lines. Fun fact: StandardJS and prettier code is not compatible by default.
These problems were also attempted to be solved on a language level, most notoriously by Python which does away with blocks through braces and uses indentation for block scoping. While it voids bikeshedding in that context, many other contexts remain.

I can’t help but think that we’re just working around the fact that our editors are just manipulating text. Or as Einstein (maybe) said:

> *A clever person solves a problem. A wise person avoids it.*

It seems rather ironic that a lot of non-bikeshedding discussion in programming is about clean separation of business logic and the user interface (also known as Model-View-?, or ???) while code itself is business logic and user interface in one. One quasi-bikeshedding prime example is the dangling comma. While the argument that with it CVS diffs tend to be cleaner certainly holds, people find it understandably visuably displeasing.

Visual programming languages to the rescue! Or not…

Fortunately we also have so called “visual programming languages”, that solve these and other problems. Specifically we shall look at the

subcategory of tree-languages, which is a term I made up to differentiate them from visual languages like LabVIEW and others that model dataflow as a graph. These graph-languages definitely have their own raison d'être (pardon my french) but I'm not yet convinced they are appropriate for general purpose programming. Anyway, back to these tree languages:

Scratch by MIT Media Lab

Widely known and quite similar are MIT's Scratch (on the left) and Google's Blockly, which transpiles to JS, Python and other languages. They present all statements and expressions as blocks that can be linked together or nested. Different colors indicate different statement types, and rely heavily on mouse interaction. Both are educational tools and not meant for production usage.

There are multiple things that make these language impractical for everyday coding tasks. I already casually mentioned the lack of keyboard support, but that might actually be the deciding reason why these tools aren't used more widely.

Another choice that strikes me as unfortunate is that all of those visual programming languages are actually their own programming

language, even though they could just as well be a structure editor over existing languages (thereby leveraging the ecosystem and allowing the editing of preexisting code). The strong separation from existing technologies plays into the binary perception of visual programming languages as just being **a** different paradigm, while in reality they are **multiple** different paradigms in one. This conflation makes it all too easy to reject it on the basis of interaction model alone. Even though their real value, and this is where I blaspheme, is in constraining what code we can write. There really should be no reason why we'd want total freedom in our code editors…

The reason we want total freedom

Code tends to change. Groups of statements are extracted into functions, expressions into variables and the other way around (also called inlining), Parameters are added, renamed, reordered, (statically) retyped and so on. Our editor needs to make these common tasks as frictionless as possible. And that's what current editors mostly succeed at. You want to add a new parameter at the first position? Just type the name and add a comma. You want to rename something? Select it and just type the new name (IDE bonus: use refactor to rename every occurence).

There are a plethora of instances where this free interaction model enables the programmer to quickly achieve what is intended. But when adherence to a consistent code style is also wanted, there is bound to be some blandness in the process of always getting it right. Nevertheless a new constrained interaction model would have to be at least as frictionless in these common interactions.

The way forward

This is what a constrained editor needs to get right to beat the current editors:

1. **Common code transformations in just as many keystrokes or less**
   Though I'm thinking less of IDE features here, as building our own refactoring system is rather complex.
   Stretch Goal: Given the rich plugin API most editors have, it might also be possible to build an editor as a Plugin that leverages these existing features.

2. **Interop with existing codebases**
   Read existing files and save them without changing lines that the user didn't touch. The aforementiond Prettier already does a

good job with that.
Stretch Goal: Read the code style of the project (e.g. .eslintrc) and apply the style rules to the lines changed by the user.

3. **Better snippets**
   Beating IntelliJ at that game should be rather difficult. Their "Live Templates" (fancy word for snippet) actually have a setting for where in the code it is applicable (e.g. Statements, Expressions, Comments, etc.). To take it a step further one could always show the language constructs that can be used in the current context. This is a good moment to mention Greenfoot. It is probably the closest tool to what I'd imagine, though it has some limits, namely it's own programming language (why oh why) and some cases of unwieldy interaction, compared to text based editors. To see what I mean, best check it out.

> *"Talk is cheap. Show me the code."*
> *- Linus Torvalds*

I started work on implementing a JavaScript version but during the process I realised, that it might be worth discussing different modes of text editing and use the wisdom of the crowd to discover edge cases which are hard to solve in a constrained editor. Basically I use the following quote as a defense against the former:

> *"You can use an eraser on the drafting table or a sledgehammer on the construction site."*
> *- Frank Lloyd Wright*

As a proof of concept I'll continue working on a version of such a constrained editor but only for JSONs, so that I don't have to deal with the entire complexity of JavaScript while still demonstrating the potential upsides of it.