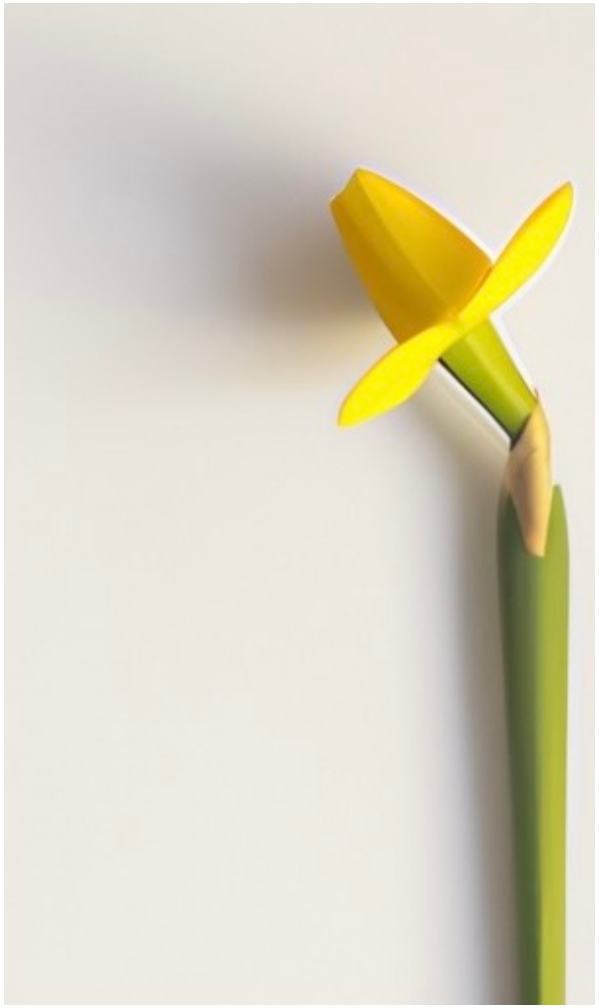
PEG - Parsing Expression Grammars



PEG Is More Flexible Than CFG

PEG Is More Flexible Than REGEX

Compiler Building Is Easier with PEG

PEG

PEG - Parsing Expression Grammars - provide a fresh way to inhale text and to exhale the text after processing and reformatting.

PEG is better at this than using CFGs, since PEG can express grammars that contain recursive, matched brackets.

PEG can express pattern-matches that include matched pairs of brackets, recursively.

PEG is a DSL for building *parsers,* whereas CFG is a DSL for expressing *languages*, with the side-effect that CFGs can also be used to build parsers, but, such parsers can't be used to build all of the parsers that PEG can build.

PEG can be used like REGEX is used, except that PEG can match structured text, whereas REGEX cannot. For example nested scopes, such as

```
int f (a) {
  b = a;
  {
    b = a + a;
  }
 }
```

Disadvantages of PEG

Many PEG libraries require that the code for pattern-matching be intermingled with the code for processing the matches (aka “semantics”). This conflation obfuscates the original grammar. See Ohm, though.

PEG specifications must, themselves, include a lot of niggly detail, like matches for noise characters, e.g. whitespace. See Ohm, though.

PEG specifications are usually longer than REGEX specs. But, many people already think that REGEXs are write-only, usually unreadable except by the creator (and only then sometimes). PEG specifications are more readable, but, longer than REGEXs.

Some syntactic bugs cannot be easily detected with PEG, like unbalanced brackets. This is similar to the *unterminated string* problem that occurs with CFG-based parsers. Can we just live with this restriction? Can both technologies be used in Production?

PEG uses backtracking. This used to be considered a wasteful practice and was *verboten* in the days of restricted hardware, e.g. in the 1950s when CPU time was expensive and memory was scarce. Today, these issues are no longer concerns, our hardware is cheaper and faster. We can use anything that makes our lives simpler.

Ohm

Ohm addresses a number of the above issues. Namely,

- Ohm separates grammar and semantics.
- Ohm grammars have a feature that removes the need for specifying handling of noise characters.

Note that the above features could be implemented in any PEG, by writing DSLs (in PEG, of course) to achieve the same programming features of Ohm.

Note that Ohm is not the *only* way to address the problems, but Ohm is a strong hint at what can be accomplished.

OhmJS is a PEG-based language that works with a popular, common underlying language - JavaScript.

OhmJS is production quality and can be used in production code and in browsers.

Using OhmJS and PEG does *not* mean having to learn how to write compilers in a heavy-handed manner.

OhmJS can be used like REGEXs are currently used, and, opens up new vistas for how to solve problems.

Language and Compiler Building With Ohm

You can write a new language in only a few hours by using Ohm as a *t2t* transpiler (text-to-text transpiler - what the goal of FP really is). Write a grammar, transpile the grammar to some existing language, let the existing compiler do the rest of the heavy lifting.

Programming Simplicity

ohmjs.org

https://guitarvydas.github.io/2024/02/05/T2T-Transpilation-To-Write-A-Compiler-or-Not-To-Write-A-Compiler.html

https://guitarvydas.github.io/2024/01/05/Macros-for-Non-Lisp-Languages.html

https://github.com/guitarvydas/arith0d

https://github.com/guitarvydas/0D

Various other repos with names ending in “0d”
https://github.com/guitarvydas/*0d

See Also

References *https://guitarvydas.github.io/2024/01/06/References.html*
Blog *https://guitarvydas.github.io/*
Blog *https://publish.obsidian.md/programmingsimplicity*
Videos *https://www.youtube.com/@programmingsimplicity2980*
[see playlist “programming simplicity”]
Discord *https://discord.gg/Jjx62ypR*
X *(Twitter) @paul_tarvydas*
More writing *(WIP): https://leanpub.com/u/paul-tarvydas*