

DPI

De Programmatica *Ipsium*

DE PROGRAMMATICA IPSUM

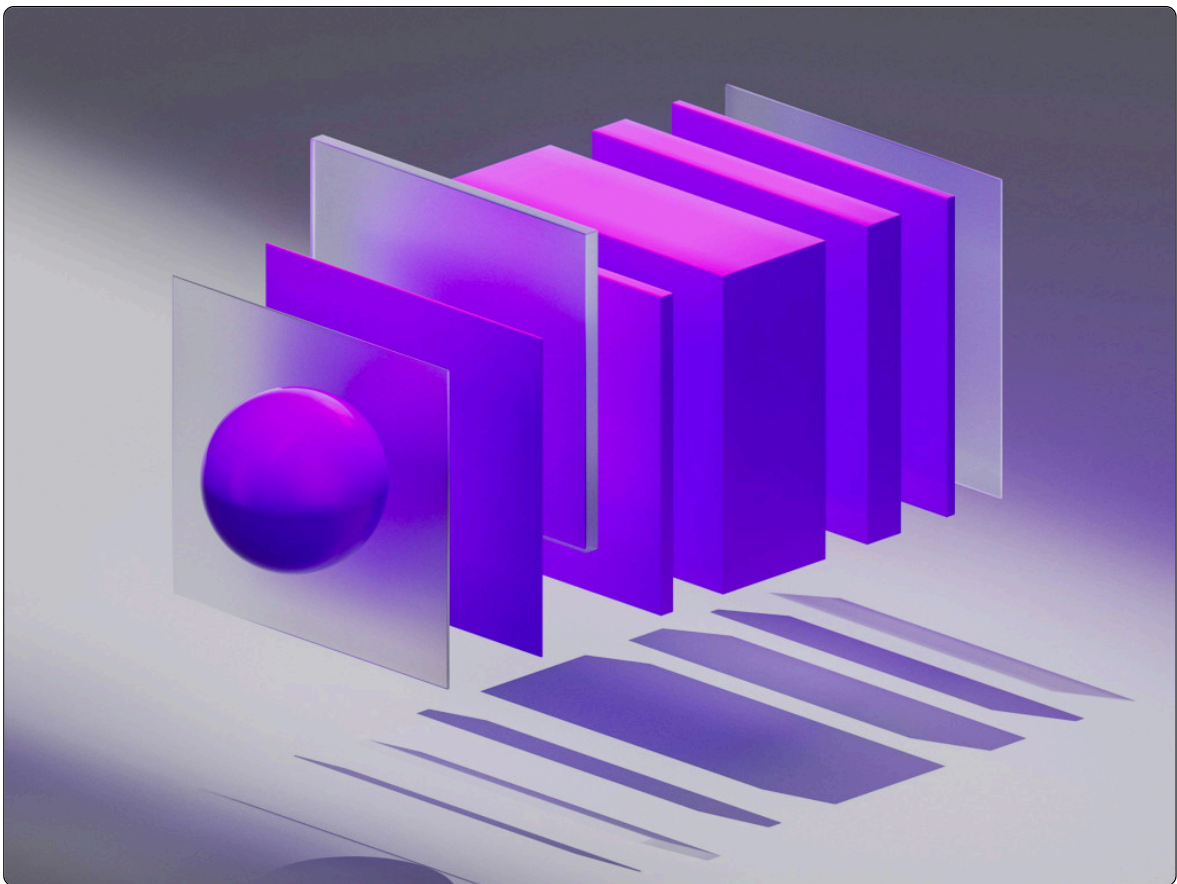
Issue 093:
Functional
Programming

June 1st, 2026

Table of Contents

Issue 093: Functional Programming	5
Evading Von Neumann	9
Functional Programming: The Good Parts	17
Joe Armstrong	27
Guy Steele & Gerry Sussman	35
Philip Wadler	41

Issue 093: Functional Programming



June 1st, 2026

Welcome to the 93rd issue of *De Programmatica Ipsum*, about *Functional Programming*.

In this edition:

- Graham explains the benefits¹ of solving problems with a functional mindset.
- Adrian explores why functional programming was shunned² until it was not.

ISSUE 093: FUNCTIONAL PROGRAMMING

- In our Vidéothèque section³, we watch Joe Armstrong⁴ explain how functional programming made Erlang a reality.
- In the Library section⁵, we review the most important papers by Philip Wadler⁶, Guy Steele, and Gerry Sussman⁷.

Download this issue in DRM-free PDF⁸ or EPUB⁹ format, and read it on your preferred device. You can also subscribe to our RSS feed¹⁰, featuring the full content of our articles.

We would like to thank our patrons who generously contribute every month (or have contributed in the past) to our work and help us run this magazine. Thank you so much! In alphabetical order: Adam Guest, Adrian Tineo Cabello, Benjamin Sheldon, Christopher Nascone, Colin Powell, Franz Lucien Moersdorf, Guillermo Ramos Álvarez, Jean-Paul de Vooght, Dr. Juande Santander-Vela, Patryk Matuszewski, Paul Hudson, Quico Moya, Roger Turner, Szymon Licau, and countless more leaving anonymous tips every month.

Enjoy this issue! Please share our articles on social media, or contribute¹¹ if you would like to support our work with a donation via Liberapay¹².

Cover photo by Milad Fakurian¹³ on Unsplash¹⁴.

REFERENCES

- ¹ <https://deprogrammaticaipsum.com/evading-von-neumann/>
- ² <https://deprogrammaticaipsum.com/functional-programming-the-good-parts/>
- ³ <https://deprogrammaticaipsum.com/category/videotheque/>
- ⁴ <https://deprogrammaticaipsum.com/joe-armstrong/>
- ⁵ <https://deprogrammaticaipsum.com/category/library/>
- ⁶ <https://deprogrammaticaipsum.com/philip-wadler/>
- ⁷ <https://deprogrammaticaipsum.com/guy-steele-gerry-sussman/>
- ⁸ <https://deprogrammaticaipsum.com/pdf/issue-093-functional-programming.pdf>
- ⁹ <https://deprogrammaticaipsum.com/epub/issue-093-functional-programming.epub>
- ¹⁰ <https://deprogrammaticaipsum.com/index.xml>
- ¹¹ <https://deprogrammaticaipsum.com/contribute/>
- ¹² <https://liberapay.com/akosma/donate>
- ¹³ https://unsplash.com/@fakurian?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText
- ¹⁴ https://unsplash.com/photos/a-purple-object-with-a-shadow-on-the-ground-Em8glt2OLt0?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

Evading Von Neumann



By Graham Lee

Programming styles are supposed to be *paradigmatic*, in that they structure your thinking about creating software by providing unifying theories and methods that you use to plan, design, construct, and operate your software. In that sense, the way that you think about the software is everything, and the tools that you use are nothing.

Take object-oriented programming¹, for example. Selecting Java as an implementation language does not immediately mean that your solution is object-oriented (though it may have been sufficient to get you some OOP-linked venture capital at

the peak of the hype cycle²). Similarly, when you see an implementation language that does not have object features, FORTRAN-77 for example, that does not mean that you are not looking at the source code of software built using the object-oriented paradigm.

But this is the issue on functional programming, not OOP. The paradigmatic basis of functional programming is that you design your program as algebraic transformations of structured data types. This affects how you discuss your program with stakeholders, how you model requirements, how you architect solutions and deployment...it is not just a matter of reaching for Haskell (indeed using the `do` form it is very easy to write procedural software in Haskell anyway).

You compose large programs by combining algebraic transformations into more complex operations. As such, you define transformations that are composable. This composability comes from designing transformations with clear contracts and interfaces, so that the result of applying the transformation is evident and safe in the context of the composition. For example, imagine that you have a need to double every element in a list of numbers. It is easy in many imperative languages to write a loop that iterates over the list and multiplies each number by 2. But thinking about this “problem” using the paradigm of composing algebraic transformations, you can build the solution from two components:

1. Apply an operation to every element of a list.
2. Multiply a number by 2.

So now you can build two functions, which are demonstrated as procedures written in Pascal³ here because, as I said, tool use is not paradigmatic; and besides, Pascal and Haskell nearly rhyme and both languages are named after mathematicians, so they are almost the same thing. The first function to create is the `Map` function, which applies a function to every element in a collection and returns a collection of the results:

```
unit Functional;  
  
interface  
  
type  
    generic TArrayU<U> = array of U;
```

```

generic TMapFunc<T, U> = function(const Item: T): U;

generic function Map<T, U>(const Arr: array of T; Func:
    specialize TMapFunc<T, U>):
    specialize TArrayU<U>;

```

implementation

```

generic function Map<T, U>(const Arr: array of T; Func:
    specialize TMapFunc<T, U>):
    specialize TArrayU<U>;
var
    i: Integer;
begin
    Result := nil;
    SetLength(Result, Length(Arr));
    for i := 0 to High(Arr) do
        Result[i] := Func(Arr[i]);
    end;
end.

```

The generic interface for the Map function means that even if you did not look at the implementation, you would know something about how it works. Its inputs are an array of T , and a function that turns T into U , and its output is an array of U . Nowhere is anything specified about T or U . That means that the function cannot use any of the operations available on either T or U , so if there is a U in the output the *only* way that Map created it was by applying the function to one of the T s in its input.

Unfortunately Pascal arrays have a rich interface so it is impossible to say anything about *which* T s get turned into U s; the function might always return `nil`, for example. The implementation shows that this Pascal Map works as you might expect a map function; it applies the `Func` parameter to each T in the input array, and returns an array where the corresponding U is at the same index as the given T .

The second function you create is the `DoubleInt` function that multiplies a number by 2 and returns the result:

```

function DoubleInt(const i: Integer): Integer;
begin
    Result := i * 2;
end;

```

Here the interface does not help much at all: the input is an `Integer`, and the result is an `Integer`. The function might always return a constant, or square the input, or add it to a constant, or convert it into a `String`, reverse the characters, and convert that back into an `Integer` to return. This demonstrates the power of generic component design: the less information available about the types used in a function, the more constraints on the behavior of that function, so the easier that function is to understand.

Now you solve the problem by composing `DoubleInt` and `Map`:

```

program Main;

uses
  Functional, SysUtils;

type
  TIntArray = array of Integer;

var
  Ints, Doubled: TIntArray;

begin
  Ints := TIntArray.Create(1, 2, 3, 4, 5);

  Write('Original: ');
  for i in Ints do Write(i, ' ');
  WriteLn;

  Doubled := specialize Map<Integer, Integer>(Ints, @DoubleInt);
  Write('Mapped (Double): ');
  for i in Doubled do Write(i, ' ');
  WriteLn;
end.

```

These two functions are very simple mappings of their input to their output. You could imagine implementing `DoubleInt` as a lookup table, where the value at index 0 is 0, the value at index 1 is 2, and so on. In principle you could do the same for `Map` but it would be a very tedious table to write, and it would lose the simplicity of using the `Func` parameter to achieve the transformation.

A problem for composability arises when you want to use a program to *do* something; to store or retrieve a result, or to interact with its environment. These

kinds of computations do not show up in the function signature, so you cannot tell the difference between “function that accepts a list of τ and returns a Boolean” and “function that accepts a list of τ , drops all tables in the database, and returns a Boolean”. It is harder to build functions into reusable workflows when they might have side effects like that.

A solution used by people who think in the functional way is monads⁴, a class of types that associate values with context and enables computations on the values while maintaining the association with the context. Again, because there is nothing special about programming language choice, you can build that in Pascal:

```

unit Monads;

interface

type
  generic IMonad<T> = interface
    ['{7A8D8C30-5A2C-4B9F-8F8D-9E9C9B9A9D9E}']
    function Execute: T;
  end;

  generic TIOReturn<T> = class(TInterfacedObject, specialize IMonad<T>)
  private
    FVal: T;
  public
    constructor Create(const V: T);
    function Execute: T;
  end;

  generic TIOBind<A, B> = class(TInterfacedObject, specialize
IMonad<B>)
  public
    type
      TBF = function(const V: A): specialize IMonad<B>;
  private
    FPrev: specialize IMonad<A>;
    FFunc: TBF;
  public
    constructor Create(const P: specialize IMonad<A>; const F: TBF);
    function Execute: B;
  end;

implementation

```

```
{...}  
end.
```

All this makes it look like there is nothing to a “functional programming language” other than the convenience of expressing paradigmatic functional thinking efficiently. For example, if you had access to a programming language that is like Pascal but also has partial application (function currying, to use Haskell Curry’s family name) then you could create the `DoubleInt` function in the first example above by applying `2` to the `Multiply` function, and avoid needing to write a custom component for such a simple operation.

Using the idea that all computationally universal⁵ models are equally powerful, you can do your functional thinking in your brain and then convert it into Haskell, Pascal, or even languages that do not rhyme and are not named after mathematicians. This statement, true as it goes, is not helpful. Functional programming as a style does benefit from specifically designed programming languages, that perform for software engineering what John Locke achieved for national government in the 17th Century: the separation of [Alonzo] Church and state.

The Pascal functions above might present interfaces that represent composable algebraic primitives, but under the hood their implementations are sequential applications of state transitions in a slightly abstracted version of the computer’s memory. Pascal’s model of computation might let it reorder some instructions a little, but fundamentally it, and similar languages like Algol, C, Rust, and Python, are there to make it easy for you to sequence changes to the computer’s memory.

A goal of functional programming is to escape the von Neumann bottleneck⁶, describing programs in ways that are not limited by the sequencing of changes to memory states. Yes, that has to happen, but it does not have to be the programmer’s responsibility. Instead, the programmer’s task is to explain what the program should do, and the *computer’s* task is to find an efficient way to do it. Tail-call recursion, lazy or applicative evaluation, memoization, parallelism—none of those are your problem.

The most widespread examples of programming tools that embody this notion of communicating intent, not computation, are query languages (and these days, coding AIs). You do not tell SQL, or LINQ, how to loop over the elements

in a collection; you tell it which properties you want from what elements (for example, composing primitives expressed in the Relational Algebra⁷ like “Select” and “Project”), and *the computer* works out an efficient way to make that happen.

You can reap the benefits of functional thinking in whatever programming language you happen to use. To truly escape the clutches of von Neumann and take advantage of functional *computation*, you do in fact need to choose your tools wisely.

Horrific disestablishment pun by Prof. Jeremy Gibbons⁸. Photo by Sanket Shah⁹ on Unsplash¹⁰.

REFERENCES

- ¹ <https://deprogrammaticaipsum.com/issue/issue-049-object-oriented-programming/>
- ² <https://deprogrammaticaipsum.com/the-hype-cycle-of-oop/>
- ³ <https://deprogrammaticaipsum.com/issue/issue-065-pascal/>
- ⁴ <https://learnyouahaskell.github.io/a-fistful-of-monads.html>
- ⁵ https://en.wikipedia.org/wiki/Turing_completeness
- ⁶ <https://dl.acm.org/doi/10.1145/359576.359579>
- ⁷ https://www.cbc.umd.edu/confcour/Spring2014/CMSC424/Relational_algebra.pdf
- ⁸ <https://www.cs.ox.ac.uk/jeremy.gibbons/>
- ⁹ https://unsplash.com/@sanketshah?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText
- ¹⁰ https://unsplash.com/photos/cooked-food-in-black-cooking-pot-eEWlcfydzQ4?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

Functional Programming: The Good Parts



By Adrian Kosmaczewski

In a famous paper¹ published in 1998, Philip Wadler² complained that no one used functional programming languages. It is safe to say that in 2026 everybody is using some kind of functional programming language, albeit to a certain extent, but the underlying reason for this spread had more to do with fashion and hype rather than market economics or academic support.

“Modern” programming languages include, almost without exception, features borrowed from the realm of functional programming languages. This can be

briefly summarized as follows: “lambdas” or “closures” that can be passed around and called *a piacere*; a series of functions to manipulate arrays called `map()`, `filter()`, and `reduce()` (provided either as standalone functions or as methods of some `Array` class provided by the languages’ runtime libraries); immutable records or classes, guaranteed to be unchanged throughout the execution of the program; and sometimes even a “pipe operator” looking like this: `|>`.

In the latter case, we can include the humble PHP³ programming language, which features⁴ such a contraption since version 8.5, released last November at the time of this writing. Which means that the two statements below produce similar results:

```
$result1 = trim(str_shuffle(strtoupper("hello")));
$result2 = strtoupper("hello") |> 'str_shuffle' |> trim(...);
```

Spoiler alert: the ellipsis (since PHP 8.1) makes a closure out of a pre-existing function, but as shown above, you can just use a string with the name of the function you would like to “pipe” your output into. Oh, and beware of the fact that `array_map()` takes the callback function as the *first* argument rather than the second, which means that... you need a wrapper to invert the arguments. Yeah, as one author describes⁵ the current situation in PHP,

The deeper issue is the stdlib itself. PHP’s stdlib was never shaped for chaining. array_map and array_filter taking arguments in different orders is a 1995 design that calcified before anyone thought about composition. The pipe operator works around the symptom. Native methods on arrays and strings would fix the cause.

(Takes a deep breath.)

Other than such small shenanigans, and thanks to the introduction of the pipe operator, recently laid-off developers coming from the F#, Elixir, OCaml, Elm, or Julia galaxies should now feel at home with PHP (well, almost). Note to Haskell developers struggling to pay rent and willing to jump on the very lucrative market of PHP: in your case the pipe operator looks like an `&`. You have been warned. Oh, and JavaScript developers will have to wait for their own pipe operator, but it might happen⁶ sooner than expected.

Let us be even more radical: according to Kevlin Henney⁷, Excel is the world's most popular functional programming language, and even better, it now comes bundled⁸ with a `LAMBDA()` function. How about that?

Are these features enough to make PHP (or Excel) a *real* functional programming language? Let us recap. Closures? Check⁹. Map, filter, and reduce functions? Check. Readonly classes? Check¹⁰. Pipe operator? As we have seen above, check.

But of course, no, PHP is not really a *pure* functional programming language. It still allows for side effects, and that is a big no-no-no in that world. Even worse, it is quite obviously a procedural language, not really a declarative one. But with a little bit of attention, and maybe with the help of your preferred LLM-powered IDE and some unit tests, your PHP code can reach levels of “functional compliance” Rasmus Lerdorf could only dream of (well, perhaps he never dreamt of such a thing; I give you that).

The real question is the following: why does a parochial programming language such as PHP now include these features? The reasons behind the choice to add them as part of the specification have a strong element of hype.

The jump to concurrency¹¹ and the economics of cloud computing¹² have given functional programming constructs a boost that neither desktop nor mainframe applications ever could. Let us be honest here: in the world of concurrency, multi-core CPUs, at the end of the “free lunch” world, functional languages shine.

The trend was apparent at the beginning of the 2000s, and it simply exploded during the 2010s. Scala was released in 2004, becoming the first functional programming language built on top of the JVM; standard Java developers would have to wait until Java 8 (released in 2014) to be able to write code including lambda expressions and its associated `map`, `filter`, and `reduce` functions. On the other side, the C# community discovered LINQ¹³ in 2005 already, effectively transforming array manipulation into a SQL-like contraption. In the galaxy of Apple, Objective-C blocks and their associated arcane syntax¹⁴ appeared together with Grand Central Dispatch¹⁵ around 2009. C++ debuted lambda expressions in its specification¹⁶ in 2011.

Languages created in the past 20 years almost inevitably include one or another of the functional programming features enumerated above: Go functions are first-class objects. TypeScript builds on top of the already existing JavaScript language. Rust closures come in various “traits” depending on how they capture their surrounding environment. Elixir brought closures on top of the Erlang BEAM. Dart has supported closures since its first version. F# brought rather pure functional programming features to the .NET runtime. Swift had first-class functions from day one, and for a while, even object-oriented methods were implemented as closures. Finally, Kotlin brought a shorter, simpler syntax for closures on top of the JVM.

Oh, and then PHP followed the trend, obviously. By the way, it is worth mentioning that Python, Perl, and Lua have had support for lambdas since the 1990s, at a time when these concepts were still considered fringe.

Undeniably, in the same vein, the rise in popularity of Python, Ruby, and JavaScript during the past two decades also contributed to the slow march of functional thinking into the minds of software developers worldwide. Speaking (again) about JavaScript, we have talked about Douglas Crockford’s *magnum opus* in a previous article¹⁷ of this magazine, and it was obvious to him that one of the “good” parts of JavaScript was, precisely, its functional programming roots:

JavaScript is built on some very good ideas and a few very bad ones.

The very good ideas include functions, loose typing, dynamic objects, and an expressive object literal notation. The bad ideas include a programming model based on global variables.

JavaScript’s functions are first class objects with (mostly) lexical scoping. JavaScript is the first lambda language to go mainstream. Deep down, JavaScript has more in common with Lisp and Scheme than with Java. It is Lisp in C’s clothing. This makes JavaScript a remarkably powerful language.

The quote above explains why, at the time of this writing, the latest version of the quintessential book on functional programming, “Structure and Interpretation

of Computer Programs”¹⁸, by Abelson and Sussman, is the “JavaScript Edition”¹⁹. The choice of JavaScript somehow gives reason to Philip Wadler²⁰, who in 1987 argued²¹ that using Scheme was probably not the best idea to begin with.

Lisp would certainly deserve a whole issue of this magazine, but chapter 5 of Waldrop’s “The Dream Machine”, a book we reviewed²² precisely a year ago, contains a beautiful description of its early history and its impact:

Nonetheless, it’s fair to say that one of Lisp’s two greatest legacies to the art of programming was a certain style, a certain exploratory approach to pushing back the software frontiers.

And the other legacy? An undeniable grace, beauty, and power. As a Lisp programmer continued to link simpler functions in to more complex ones, he or she would eventually reach a point in where the whole program was a function—which, of course, would also be just another list. So to execute that program, the programmer would simply give a command for the list to evaluate itself in the context of all the definitions that had gone before. And in a truly spectacular exercise in self-reference, it would do precisely that. In effect, such a list provided the purest possible embodiment of John von Neumann’s original conception of a stored program: it was both data and executable code, at one and the same time.

Since its presentation in a now legendary paper by John McCarthy²³, Lisp’s power and beauty have often been mentioned by pundits and entrepreneurs alike. In the latter group, we cannot avoid mentioning Paul Graham: his 2001 article “Beating the Averages”²⁴ tells the story of his 1995 e-commerce startup “Viaweb”²⁵, built in Lisp, a language he deeply²⁶ loved²⁷. That article also makes a surprising connection between Lisp and the (badly named) “Sapir-Whorf hypothesis” of linguistic relativity²⁸:

By induction, the only programmers in a position to see all the differences in power between the various languages are those who understand the most powerful one. (This is probably what Eric Raymond meant about Lisp making you a better programmer.) You can’t trust the opinions of the others, because of

the Blub paradox: they're satisfied with whatever language they happen to use, because it dictates the way they think about programs.

Paul Graham was not the first to make the connection to the Sapir-Whorf hypothesis, by the way: Kenneth Iverson²⁹, creator of the APL³⁰ programming language, also explained this during his 1980 Turing Award lecture, “Notation as a Tool of Thought”³¹ as follows:

Concerning language, George Boole in his Laws of Thought asserted “That language is an instrument of human reason,, and not merely a medium for the expression of thought, is a truth generally admitted.”

Mathematical notation provides perhaps the best-known and best-developed example of language used consciously as a tool of thought.

Those of us brave enough to have written code in APL can only agree that its notation brings a whole new meaning to the word “programming”.

So it was that, standing on the shoulder of this giant called Lisp, people came up with projects ranging from Medley Interlisp³² to a Lisp interpreter built as Conway's Game of Life³³.

Lisp, and later functional programming languages, have enlightened software programmers to go above and beyond what commercial offerings could even dream of. They have stretched our thinking, opening doors to concurrency, mathematical breakthroughs, early artificial intelligence efforts, and the widest possible array of scientific exploration about computation.

Yet, already in the 1990s it was apparent that Lisp (and, to a large extent, functional programming) was at risk³⁴ of being relegated or even of becoming extinct, and today some influencers ask themselves³⁵, where did all the functional programmers go? The answer is simple: they are all among us; we are all, to a small or large degree, better software engineers because there has been a healthy breeding process between procedural, object-oriented, and functional programming languages that begat modern tools for a modern world.

All things considered, the most important contribution of functional languages in the modern world was a *certain way of thinking* about programming, more than just actually adopting “pure” functional compilers in our day-to-day jobs. Put in other words, mastering functional programming concepts fundamentally transformed the way we create software. And in the best possible way.

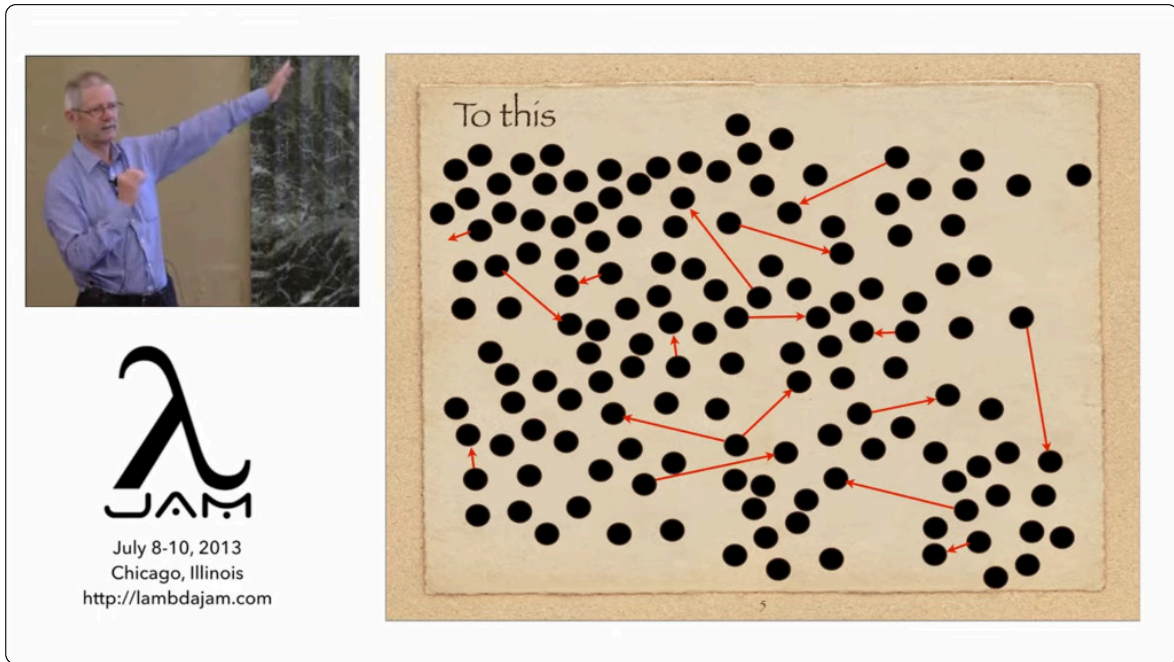
Cover photo by Pablo Martinez³⁶ on Unsplash³⁷.

REFERENCES

- ¹ <https://dl.acm.org/doi/10.1145/286385.286387>
- ² <https://deprogrammaticaipsum.com/philip-wadler/>
- ³ <https://deprogrammaticaipsum.com/the-toyota-corolla-of-programming/>
- ⁴ <https://php.watch/versions/8.5/pipe-operator>
- ⁵ <https://dev.to/delacry/php-85s-pipe-operator-hits-a-wall-on-array-code-5c7e>
- ⁶ <https://github.com/tc39/proposal-pipeline-operator>
- ⁷ <https://www.youtube.com/shorts/YPSGL2MsES4>
- ⁸ <https://www.youtube.com/watch?v=Rm4y5UqauRw>
- ⁹ <https://www.php.net/manual/en/class.closure.php>
- ¹⁰ https://wiki.php.net/rfc/readonly_classes
- ¹¹ <https://deprogrammaticaipsum.com/the-age-of-concurrency/>
- ¹² <https://deprogrammaticaipsum.com/somebody-elses-computer-as-a-service/>
- ¹³ <https://learn.microsoft.com/en-us/dotnet/csharp/linq/>
- ¹⁴ <https://fuckingblocksyntax.com/>
- ¹⁵ https://en.wikipedia.org/wiki/Grand_Central_Dispatch
- ¹⁶ <https://en.cppreference.com/cpp/language/lambda>
- ¹⁷ <https://deprogrammaticaipsum.com/douglas-crockford/>
- ¹⁸ https://en.wikipedia.org/wiki/Structure_and_Interpretation_of_Computer_Programs
- ¹⁹ <https://mitpress.mit.edu/9780262543231/structure-and-interpretation-of-computer-programs/>
- ²⁰ <https://deprogrammaticaipsum.com/philip-wadler/>
- ²¹ <https://dl.acm.org/doi/10.1145/24697.24706>
- ²² <https://deprogrammaticaipsum.com/j-c-r-licklider-m-mitchell-waldrop/>
- ²³ <https://dl.acm.org/doi/10.1145/367177.367199>
- ²⁴ <https://paulgraham.com/avg.html>
- ²⁵ <https://en.wikipedia.org/wiki/Viaweb>
- ²⁶ <https://www.paulgraham.com/onlisp.html>
- ²⁷ <https://www.paulgraham.com/diff.html>
- ²⁸ https://en.wikipedia.org/wiki/Linguistic_relativity
- ²⁹ https://amturing.acm.org/award_winners/iverson_9147499.cfm
- ³⁰ [https://en.wikipedia.org/wiki/APL_\(programming_language\)](https://en.wikipedia.org/wiki/APL_(programming_language))
- ³¹ <https://dl.acm.org/doi/10.1145/358896.358899>
- ³² <https://interlisp.org/>
- ³³ <https://woodrush.github.io/blog/posts/2022-01-12-lisp-in-life.html>
- ³⁴ <https://dl.acm.org/doi/abs/10.1145/192590.192600>
- ³⁵ <https://www.youtube.com/watch?v=gE6nnDsh5Ck>
- ³⁶ https://unsplash.com/@pablomp?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

³⁷ https://unsplash.com/photos/silver-and-red-metal-tools-S1xZ5GmpdM8?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

Joe Armstrong



By Adrian Kosmaczewski

If there was a contest for the single most beloved person in the functional programming galaxy, Joe Armstrong¹ would have effortlessly won the first prize. For decades, he constantly showed the world that the principles behind functional programming were the key for resilient, concurrent, and highly available systems. And he showed it in the best possible way, which most probably made Pastor Manul Laphroaig² very proud: with an astonishingly serious “PoC” called Erlang³.

That is why we have chosen as this month’s Vidéotheque choice his presentation titled “Systems that run forever self-heal and scale”⁴ at the 2013 Lambda Jam⁵, a conference that featured an impressive schedule⁶ with talks by Ola Bini⁷, Chris Ford⁸, Dave Thomas⁹ (not the Pragmatic Dave¹⁰), Bartosz Milewski¹¹, Adam Granicz¹², Steve Vinoski¹³, and Gerald Jay Sussman¹⁴.

As you most probably know by now, Joe Armstrong (1950-2019) is mostly known for his work on Erlang, a massively concurrent and functional programming language, coupled with BEAM, the virtual machine that enabled telcos, starting in the 1990s, to serve millions of users with complex systems in the most efficient and resilient way.

Kids, this was 20 years before Go¹⁵ or Kubernetes¹⁶ were even conceived. Erlang enabled companies to scale their services in ways that were unthinkable even by today's standards, running on hardware way underpowered compared to that of our modern world of 2026.

Joe kicked off his acclaimed book “Programming Erlang” (released in 2007 by The Pragmatic Programmers, with a second edition¹⁷ published in 2013) with a clear disclaimer:

In many places we'll be extolling the virtues of functional programming. Functional programming forbids code with side effects. Side effects and concurrency don't mix. You can have sequential code with side effects, or you can have code and concurrency that is free from side effects. You have to choose. There is no middle way.

And he goes on to explain on page 44 (mind you, this was written in 2007) what “functional” means in this context:

Erlang is a functional programming language. Among other things this means that funs can be used as arguments to functions and that functions (or funs) can return funs.

Functions that return funs, or functions that can accept funs as their arguments, are called higher-order functions. We'll see a few examples of these in the next sections.

I stress the fact that this was written in 2007 because merely 20 years ago, as we were entering the “plateau of productivity” in the hype cycle¹⁸ of Object-Oriented Programming, we were also witnessing how Twitter (the original name of a

decadent social network still active as this article hits the press) was suffering with “fail whales” shown on its home page, while millions of users were trying to read the tweets on their (not yet algorithm-driven) feeds. The time was ripe for a new paradigm... but apparently, unbeknownst to the Twitter team, it already existed.

Interestingly, the WhatsApp developer team knew exactly that they needed something else to create a system potentially usable by billions of simultaneous users, so they chose Erlang, and boy did that work well. More on that later.

In his 2013 conference talk, Joe Armstrong started with some simple observations: the real world is parallel. Boom. It turns out that Erlang processes are the perfect way to model such a world: they can be thought of as a group of people communicating by message passing.

Let us remember what Alan Kay said¹⁹ about messaging in 1998:

The big idea is “messaging” – that is what the kernel of Smalltalk/Squeak is all about (and it’s something that was never quite completed in our Xerox PARC phase). The Japanese have a small word – ma – for “that which is in between” – perhaps the nearest English equivalent is “interstitial”. The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be.

So here we have a programming language that allows you to model a world with certain characteristics that Joe describes around minute 05:30²⁰: many computers distributed all over the place, working concurrently, detecting their own failures and repairing them as soon as possible, and even featuring a radical concept called live code upgrades.

Sounds familiar? Any parallels with Kubernetes are just a coincidence. In Erlang, there is no such thing as an atomic update of the “stop it, upgrade, restart” kind (06:50²¹): Erlang applications are continuously partially upgrading themselves whenever needed.

Needless to say, this was beyond impressive in the mid-1990s, but Sun²² had a bigger marketing budget than Ericsson. Insert sad face emoticon here.

Erlang was designed from the ground up for “5 nines reliability” (07:10²³) because of a simple observation: it is much better to design a system for 10 million users and scale it down to 10,000 than to scale it up from 10 to 10,000. The result is that by 2013 there was a 50% chance that smartphones went through Erlang to talk to the mobile Internet.

But Erlang is just a piece of the whole architectural cake, albeit a critical one. Joe goes on to elaborate on the patterns required for system consistency and fault tolerance, distributed consensus (23:36²⁴), and the evolution from Lamport’s Paxos²⁵ to Ongaro’s and Ousterhout’s Raft²⁶. He also mentions six rules for fault tolerance (30:00²⁷), applicable to any massively distributed system running on your nearest Kubernetes cluster nowadays: process isolation, concurrency, failure detection, fault identification, live code upgrade, and stable storage.

Legendary computer scientist Jim Gray²⁸ wrote a widely quoted paper in 1986 titled “Why Do Computers Stop and What Can Be Done About It?”²⁹ where he said:

The top priority for improving system availability is to reduce administrative mistakes by making self-configured systems with minimal maintenance and minimal operator interaction.

(...)

As with hardware, the key to software fault-tolerance is to hierarchically decompose large systems into modules, each module being a unit of service and a unit of failure. A failure of a module does not propagate beyond the module.

Now you understand why your Kubernetes Deployment YAML contains a readiness and a liveness probe, for example. You are welcome.

Towards minute 46³⁰ of the video, Joe goes on to talk about Erlang in detail, enumerating some success stories: Mnesia³¹, CouchDB³², Riak³³, ejabberd³⁴, RabbitMQ³⁵, and, yes, of course, WhatsApp, snapped up by Facebook for a hefty 20 billion dollars of pocket money. And then he goes on to discuss the future of Erlang as seen from the perspective of 2013: Elixir³⁶, a programming language based on BEAM and OTP but offering an admittedly much friendlier syntax (inspired by Ruby³⁷) with interesting metaprogramming possibilities.

Watch this month's Vidéotheque entry, "Systems that run forever self-heal and scale," by Joe Armstrong, on YouTube³⁸. Continue binge-watching "Erlang: The Movie," a 1990 short nowadays available on YouTube³⁹ and the Internet Archive⁴⁰, showing a demo of a bug-fixing session on a live Erlang system. Let us repeat for the people in the back: 1990⁴¹.

Cover snapshot chosen by the author.

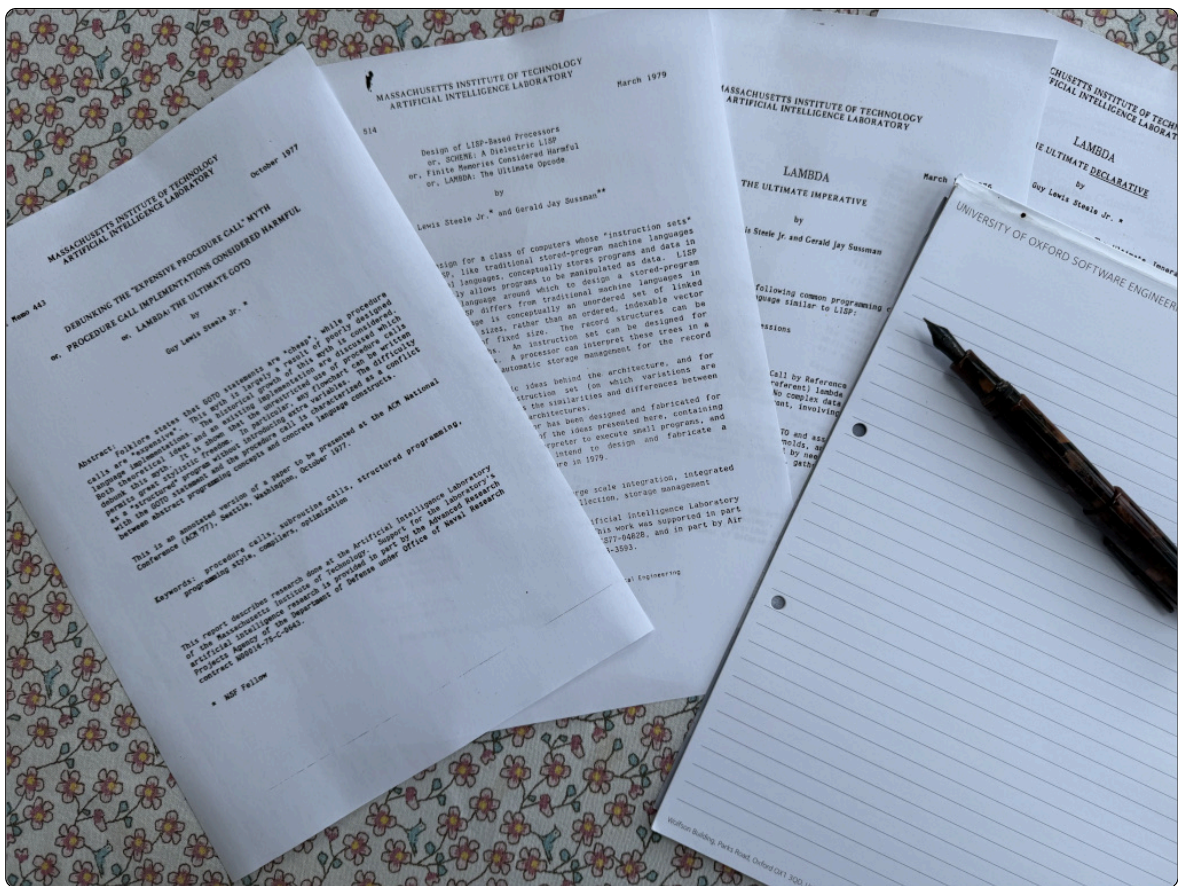
REFERENCES

- ¹ [https://en.wikipedia.org/wiki/Joe_Armstrong_\(programmer\)](https://en.wikipedia.org/wiki/Joe_Armstrong_(programmer))
- ² <https://deprogrammaticaipsum.com/pastor-manul-laphroaig/>
- ³ [https://en.wikipedia.org/wiki/Erlang_\(programming_language\)](https://en.wikipedia.org/wiki/Erlang_(programming_language))
- ⁴ <https://www.youtube.com/watch?v=cNICGEwmXLU>
- ⁵ <https://web.archive.org/web/20130603195022/http://lambdajam.com/>
- ⁶ <https://web.archive.org/web/20130623055057/http://lambdajam.com/schedule/>
- ⁷ https://en.wikipedia.org/wiki/Ola_Bini
- ⁸ <https://www.thoughtworks.com/profiles/c/chris-ford>
- ⁹ https://www.davethomas.net/biography_index.html
- ¹⁰ [https://en.wikipedia.org/wiki/Dave_Thomas_\(programmer\)](https://en.wikipedia.org/wiki/Dave_Thomas_(programmer))
- ¹¹ <https://bartoszmilewski.com/>
- ¹² <https://scholar.google.com/citations?user=k73mdkkAAAAJ&hl=en>
- ¹³ <https://www.informit.com/articles/article.aspx?p=1399235>
- ¹⁴ https://en.wikipedia.org/wiki/Gerald_Jay_Sussman
- ¹⁵ <https://deprogrammaticaipsum.com/the-age-of-concurrency/>
- ¹⁶ <https://deprogrammaticaipsum.com/antonomasia/>
- ¹⁷ <https://pragprog.com/titles/jaerlang2/programming-erlang-2nd-edition/>
- ¹⁸ <https://deprogrammaticaipsum.com/the-hype-cycle-of-oop/>
- ¹⁹ <https://lists.squeakfoundation.org/pipermail/squeak-dev/1998-October/017019.html>
- ²⁰ <https://youtu.be/cNICGEwmXLU?t=330>
- ²¹ <https://youtu.be/cNICGEwmXLU?t=409>
- ²² <https://deprogrammaticaipsum.com/issue/issue-024-java/>
- ²³ <https://youtu.be/cNICGEwmXLU?t=429>
- ²⁴ <https://youtu.be/cNICGEwmXLU?t=1416>
- ²⁵ [https://en.wikipedia.org/wiki/Paxos_\(computer_science\)](https://en.wikipedia.org/wiki/Paxos_(computer_science))
- ²⁶ [https://en.wikipedia.org/wiki/Raft_\(algorithm\)](https://en.wikipedia.org/wiki/Raft_(algorithm))
- ²⁷ <https://youtu.be/cNICGEwmXLU?t=1769>
- ²⁸ [https://en.wikipedia.org/wiki/Jim_Gray_\(computer_scientist\)](https://en.wikipedia.org/wiki/Jim_Gray_(computer_scientist))
- ²⁹ <https://pages.cs.wisc.edu/~remzi/Courses/739/Fall2018/Papers/gray85-easy.pdf>
- ³⁰ <https://youtu.be/cNICGEwmXLU?t=2749>
- ³¹ <https://en.wikipedia.org/wiki/Mnesia>
- ³² https://en.wikipedia.org/wiki/Apache_CouchDB
- ³³ <https://en.wikipedia.org/wiki/Riak>
- ³⁴ <https://en.wikipedia.org/wiki/Ejabberd>
- ³⁵ <https://en.wikipedia.org/wiki/RabbitMQ>
- ³⁶ [https://en.wikipedia.org/wiki/Elixir_\(programming_language\)](https://en.wikipedia.org/wiki/Elixir_(programming_language))
- ³⁷ <https://deprogrammaticaipsum.com/issue/issue-089-ruby/>
- ³⁸ <https://www.youtube.com/watch?v=cNICGEwmXLU>
- ³⁹ <https://www.youtube.com/watch?v=xrIjfljssLE>

⁴⁰ <https://archive.org/details/ErlangTheMovie>

⁴¹ <http://lambda-the-ultimate.org/node/197>

Guy Steele & Gerry Sussman



By Graham Lee

Imagine a world in which two people take the best ideas from programming languages, and create an interpreter for their own programming language. Then they demonstrate that most of the features in that programming language—indeed, in *all* programming languages—can be constructed out of just three features of their interpreter: lambda application, conditional execution, and variable assignment. Then, they show that variable assignment is the wrong way to think about variable

assignment, and show that their interpreter points to the most efficient way to make language compilers, and made a compiler for their interpreted language to show how good that could be. Then, imagine that they share this knowledge with the world, for free, through a series of memos.

That world that you just imagined? We live in it. Meet Guy L. Steele, Jr., and Gerald J. Sussman, two of the luminary thinkers from the Massachusetts Institute of Technology Artificial Intelligence Lab. This is the first go around the AI hype cycle, back when computers were routinely called thinking machines¹ but before people even pretended that computers were doing any thinking².

However, it is really not that valuable to distinguish between AI computing and non-AI computing, because all computation is an emulation of intelligence³. The AI research community is best thought of as an advanced computational techniques community that began life exploring computational methods to understand thought, and that is where this story begins. The person who coined the phrase “Artificial Intelligence” was John McCarthy, and for no more highbrow reason than that he thought if he presented his work in symbolic computation techniques as “cybernetics” then he would end up having an argument with Norbert Wiener, and if he called it “information processing” then the argument would be with Claude Shannon. Both of these intellectual giants were too scary to argue with.

McCarthy consulted for IBM on the addition of a List Processing library to FORTRAN⁴, but preferred to use an algebra based on symbolic expressions so created his own programming language, LISP. For the first few years LISP users had to execute their programs by hand using pencil and paper, until Steve Russell noticed a huge opportunity. The language includes a neat trick called `eval` which interprets a LISP expression as a LISP program, and Russell realised he could implement `eval` in machine language (by punching holes into punchcards) on the IBM 704. Thus it was, with the creator of LISP and the creator of AI research being the same person, that LISP became the preferred programming language of AI researchers.

Fast-forward to the mid-1970s, and the MIT AI Lab (counting McCarthy among its alumni, of course) used a LISP dialect called `Maclisp`: nothing to do with the

later personal computer model, but named after the Project on Mathematics And Computation that hired the original AI group (including McCarthy, of course). Maclisp's innovation over LISP is the use of value cells⁵ to associate objects with symbols, where LISP maintained a list of associations that it scanned through to find the object. Two of the AI lab members—the heroes of this story—wanted to explore the Actor model⁶ of computation (another product of the AI research galaxy), and did so in their own, minimalist LISP interpreter, which they wrote (inevitably) in Maclisp.

They borrowed a neat idea from the ALGOL programming language: block structure, and lexical scope. A variable (or function, or label) can be declared inside a block, in which case it is *local* to the block: it only exists while that block's in scope. If the variable has the same name as an existing variable from another scope, it *shadows* that variable, replacing uses of variables of that shared name with itself. But only within the block.

Steele and Sussman documented this Lisp interpreter, which they called Scheme⁷, through a series of AI Memos now sometimes called the LAMBDA papers⁸, after the recurring form “LAMBDA: the Ultimate x” in their titles. In LAMBDA: the Ultimate Imperative⁹, they show that lambda application can model almost every feature of an imperative language (using ALGOL in their case, but the same would apply to FORTRAN, C, Swift, Rust, or your favourite poison). Steele then issued a correction a few months later, LAMBDA: the Ultimate Declarative¹⁰.

Whoops! Did we say that lambda was the ultimate imperative? What we meant was that function calling is the ultimate imperative if you think of a function call as a GO TO statement with a message alongside it (sending messages? That is the actor model achieved!); what lambda gives you is a way to rename variables, defining an environment in which your fancy GO TO operates. That is a powerful idea in itself, because it means you do not have to mess around copying values into registers or onto the stack whenever you call a function; you just associate the new name with the existing value. He further explored this idea in Lambda: the Ultimate GOTO¹¹.

Eventually, in 1979, the two authors published LAMBDA: the Ultimate Opcode¹², in which they design the LISP machine: a hardware implementation of a state ma-

chine that evaluates LISP expressions, along with operations that work efficiently with the linked data structures native to the language.

Reading this series of AI memos in 2026, which are written in a straightforward, tutorial style, gives one first the feeling that understanding the complexity of computers and programming languages might not be so difficult, after all. Then, one remembers that their current problem involves multiple programming languages and script files and YAML files and TOML files, and the effect is more like being Charlton Heston looking up at the Statue of Liberty.

Cover photo by the author.

REFERENCES

- ¹ <https://www.sciencemuseum.org.uk/objects-and-stories/thinking-machines-stories-history-computing>
- ² <https://machinelearning.apple.com/research/illusion-of-thinking>
- ³ <https://www.sicpers.info/2026/03/on-thinking-machines/>
- ⁴ <https://dl.acm.org/doi/10.1145/800025.1198360>
- ⁵ https://softwarepreservation.computerhistory.org/LISP/MIT/Moon-MACLISP_Reference_Manual-Apr_08_1974.pdf
- ⁶ <https://dl.acm.org/doi/abs/10.5555/1624775.1624804>
- ⁷ <https://www.scheme.org>
- ⁸ <https://research.scheme.org/lambda-papers/>
- ⁹ <https://dspace.mit.edu/entities/publication/a40a9ba3-619a-4495-b357-5c2eb2442066>
- ¹⁰ <https://dspace.mit.edu/entities/publication/4b273252-5dbe-488f-891b-0c63eaaf25a1>
- ¹¹ <https://dspace.mit.edu/entities/publication/77b3eea1-50be-4e3f-b578-264484ee0f1a>
- ¹² <https://dspace.mit.edu/entities/publication/e70f4d48-b51c-4b29-996f-44ecb22441a4>

Philip Wadler



By Adrian Kosmaczewski

On page 138 of the second edition of his 1971 book, “Categories for the Working Mathematician”¹, American mathematician Saunders Mac Lane² inadvertently coined one of the most famous memes³ ever made around programming. It is there, precisely there, and not anywhere else, where the phrase “*a monad in X is just a monoid in the category of endofunctors*” was published for the first time. As is often the case, the true origin of the meme got lost in collective memory, and it ended up being falsely attributed to Philip Wadler⁴, although, in hindsight and all things considered, it was an understandable oversight.

The source of the confusion is none other than an excerpt of a masterpiece often quoted⁵ in the pages of this magazine: James Iry’s 2009 brilliant “A Brief, Incomplete, and Mostly Wrong History of Programming Languages”⁶:

1990 - A committee formed by Simon Peyton-Jones, Paul Hudak, Philip Wadler, Ashton Kutcher, and People for the Ethical Treatment of Animals creates Haskell, a pure, non-strict, functional language. Haskell gets some resistance due to the complexity of using monads to control side effects. Wadler tries to appease critics by explaining that “a monad is a monoid in the category of endofunctors, what’s the problem?”

The jury is still out debating the role that Ashton Kutcher had in the creation of Haskell, but as usual, I digress.

What is important to consider is that, yes, Philip Wadler might as well have coined the “endofunctors” meme; after all, he has singlehandedly done more to popularize and to extend the usage of functional programming languages than anyone else in the history of computing. And he has even coined the term “list comprehension”, which might ring a bell among Python developers, among others.

On the other hand, monads are a relatively recent addition to the arsenal of functional programmers; Italian computer science professor Eugenio Moggi⁷ was the first to describe the use of monads in programming in two papers published between 1989⁸ and 1991⁹. Philip Wadler took the idea, bolted it down into Haskell, James Iry wrote his blog post, and the Internet did the rest¹⁰.

I am kidding; Wadler did more than just take the idea: he helped everyone comprehend¹¹ how monads could be used to marry the “impure” necessities of actual programming with the “pure” realm of functional programming:

Purity has its regrets, and all programmers in pure functional languages will recall some moment when an impure feature has tempted them. For instance, if a counter is required to generate unique names, then an assignable variable seems just the ticket. In such cases it is always possible to mimic the required impure feature by straightforward though tedious means. For instance, a counter

can be simulated by modifying the relevant functions to accept an additional parameter (the counter's current value) and return an additional result (the counter's updated value).

A quick search shows that Mr. Wadler had a strong interest in such a marriage early on; case in point, his 1985 paper “How to replace failure by a list of successes: a method for exception handling, backtracking, and pattern matching in lazy functional languages”¹² where he argues that

The method itself is straightforward. Each term that may raise an exception or backtrack is replaced by a term that returns a list of values.

Mr. Wadler went above and beyond, explaining that monads were “the essence of functional programming”¹³ (1992) and then using them to perform input/output in a pure functional way in the widely acclaimed 1993 paper “Imperative functional programming”¹⁴ (winner of the 2003 ACM SIGPLAN Most Influential POPL Paper Award¹⁵):

We need a way to reconcile being with doing: an expression in a functional language denotes a value, while an I/O command should perform an action. We integrate these worlds by providing a type $IO\ a$ denoting actions that, when performed, may do some I/O and then return a value of type a .

Speaking about “monads” and “functions” and whatnot, the astute reader will remark that mostly everything that has to do with functional programming is related, eventually, to some obscure mathematical¹⁶ theory most self-taught software developers (like the one writing these words) have never heard about, created by a long list of mathematicians including Peano, Cantor, Russell¹⁷, and Gödel¹⁸.

And let us not forget the list of functional programming languages whose names derive from mathematicians: Haskell, Curry¹⁹, Church²⁰, Gödel²¹, Russell²², Frege²³, Rocq²⁴ (originally named “Coq” after French mathematician Thierry Coquand²⁵), Kleenex²⁶ (named after American mathematician Stephen Cole Kleene²⁷), and Erlang²⁸. Honorific mentions of the non-functional languages

Euler²⁹, Pascal³⁰, Napier⁸⁸³¹, occam³², Z notation³³ (whose name derives from Ernst Zermelo³⁴), and Ada are definitely worth mentioning.

Wait, what is Russell's name doing in the previous lists? Well, according to a 2003 paper³⁵ by Kevin Klement, from the University of Massachusetts, unpublished papers from Russell suggest that he did much more than just suggest the notation that Alonzo Church used for lambda calculus. But we leave this bit to the reader to explore.

A certain D. A. Turner, from the University of Kent & Middlesex University, wrote in "Some History of Functional Programming Languages"³⁶ (2013) that the lineage of functional programming starts in the 1930s with Church & Rosser's Lambda Calculus (a very mathematical theory of computation), then flows through McCarthy's LISP, Algol 60 (believe it or not), and Gordon & Milner's ML, culminating in the creation of Haskell from 1987 onwards:

In what follows I have, firstly, focussed on the developments leading to lazy, higher order, polymorphically typed, purely functional programming languages of which Haskell is the best known current example.

Would we be too picky to argue that Mr. Turner forgot to include John Backus, of FORTRAN and ALGOL fame, who in his 1977 Turing Award³⁷ lecture proposed³⁸ functional programming as a viable alternative to the Von Neumann programming model?

In this section we give an informal description of a class of simple applicative programming systems called functional programming (FP) systems, in which "programs" are simply functions without variables. The description is followed by some examples and by a discussion of various properties of FP systems.

Philip Wadler is a member of the original committee behind Haskell, probably the most successful exception in the world of committee-created things. Not only that, he also co-created this thing called XQuery (which you might have used 20 years ago to parse some XML, a format not so much *en vogue* these days) and was also

heavily involved in bolting generics on top of Java 5 (a language and a feature that, yes, you might have had a much higher probability of using).

Haskell enjoys quite a level of popularity in our century, similar to that of Lisp in previous decades. Even Edsger Dijkstra defended it³⁹ in 2001, when a group at the University of Austin, Texas, decided to replace Haskell with Java in the introductory programming course:

A very practical reason for preferring functional programming in a freshman course is that most students already have a certain familiarity with imperative programming. Facing them with the novelty of functional programming immediately drives home the message that there is more to programming than they thought.

But again, Haskell is a subject in itself, and I cannot help but digress once again.

As explained in the “Frequently Asked Questions for comp.lang.functional”⁴⁰ page, Philip Wadler collaborated in 1997 with a certain Martin Odersky to create “Pizza”⁴¹, a functional superset of Java... any similarities with Scala are, I suppose, just a coincidence.

Oh, and Mr. Wadler is a prolific writer of research papers and books⁴², among which stand out the chapters he contributed to “The Implementation of Functional Programming Languages” (1987) by his aforementioned partner in crime, Simon Peyton-Jones (and available for free⁴³ on the Microsoft Research website at the time of this writing). Also notable is “Introduction to Functional Programming”⁴⁴ (1988), co-authored with Richard Bird and renamed “Introduction to Functional Programming using Haskell” for its second edition published in 1998.

Among his most famous papers, let us mention “Theorems for Free!”⁴⁵ (1989), “Call-by-Value is Dual to Call-by-Name”⁴⁶ (2003), “The essence of functional programming”⁴⁷ (1992), and “A History of Haskell: Being Lazy With Class”⁴⁸ (2007), the latter being a very appropriate entry for the upcoming HOPL V conference in 2035. Furthermore, a full-text search for his family name on the Journal of Functional Programming (of which Wadler was editor from 1990 to 2004) returns⁴⁹ a staggering 421 entries at the time of this writing.

All this record of writing and conference talks⁵⁰ is quite opinionated. Philip Wadler has made his life's mission to show that functional programming languages could be used in the real world as a counterstrike against the overwhelming marketing campaigns around those languages that are not. In his "An Angry Half-Dozen"⁵¹ paper (1998), he describes 6 major technological wonders brought to life thanks to functional programming, and one of them is Erlang:

Erlang bears a striking resemblance to another modern phenomenon, Java. Like Java, Erlang (along with all other functional languages) uses heap allocation and garbage collection, and ensures safe execution that never corrupts memory. Like Java, Erlang comes with a library that provides functionality independent of a particular operating system. Like Java, Erlang compiles to a virtual machine, ensuring portability across a wide range of architectures. And like Java, Erlang achieved its first success based on interpreters for the virtual machine, with faster compilers coming along later.

That same year, he tried to explain why no one was using functional languages⁵²:

If a manager chooses to use a functional language for a project and the project fails, then he or she will certainly be fired. If a manager chooses C++ and the project fails, then he or she has the defense that the same thing has happened to everyone else.

Ouch. The following year he repeated himself, merging the previous two papers into another one called "How enterprises use functional languages, and why they don't"⁵³. Was anybody listening at all?

Philip Wadler has kept a record of every important use of functional programming on his own website since at least 1999, as the Internet Archive can testify⁵⁴, and this effort continues⁵⁵ at the time of publication of this article.

However, and rather surprisingly, Pandoc⁵⁶ is not mentioned in this list, and yet, it is literally made with Haskell! We are delighted to use Pandoc every month to produce the beautiful (DRM-free, by the way) PDF and EPUB files that you can download from this website.

I hope the core idea of this article is clear. Philip Wadler is a towering⁵⁷ figure in the field of functional programming, and the pervasiveness of functional features in most mainstream programming languages nowadays owes a lot to the efforts of this single person.

Haskell is a subject in and of itself, and there have been an incredible number of excellent books written about it in the past two decades. At the risk of leaving aside some obviously outstanding entries, we should mention at least “Real World Haskell”⁵⁸ by Bryan O’Sullivan, Don Stewart, and John Goerzen; “Learn You a Haskell for Great Good!”⁵⁹ by Miran Lipovača (this one taking more than a few cues from `_why`⁶⁰’s “Poignant Guide to Ruby”); and the more recent “Haskell Programming from First Principles”⁶¹ by Christopher Allen and Julie Moronuki.

However, if you are just interested in a rather short and high-level overview of the Haskell language, “Seven Languages in Seven Weeks”⁶² by Bruce Tate (2010) will give you an excellent starting point for your exploration. Precisely, Chapter 8, dedicated to Haskell, starts like this:

Haskell represents purity and freedom for many functional programming purists. It’s rich and powerful, but the power comes at a price. You can’t eat just a couple of bites. Haskell will force you to eat the whole functional programming burrito.

You have been warned.

To close this article, we feel that the general functional programming literature is so vast, we can only scratch the surface by enumerating some hallmark titles; please do not be offended if your favorite title does not appear on this list:

- “The Implementation of Functional Programming Languages”⁶³, by Simon Peyton Jones and Philip Wadler.
- “Introduction to Functional Programming”⁶⁴, by Richard Bird and Philip Wadler.
- “Structure and Interpretation of Computer Programs”⁶⁵, in either edition, in Scheme or in JavaScript, by Harold Abelson and Gerald Jay Sussman.
- “The Genius of Lisp”⁶⁶, a recently released gem by Cees de Groot.

- “Practical Common Lisp”⁶⁷, a classic book by Peter Seibel.
- “Paradigms of Artificial Intelligence Programming”⁶⁸, another historically relevant book by Peter Norvig.
- “Introduction to Lambda Calculus”⁶⁹, a freely available book by Henk Barendregt and Erik Barendsen.
- “From Mathematics to Generic Programming”⁷⁰, by Alexander Stepanov & Daniel Rose.
- “F# for Fun and Profit”⁷¹ and “Domain Modeling Made Functional”⁷², both by Scott Wlaschin.
- “Functional Swift”⁷³, by Chris Eidhof, Florian Kugler, and Wouter Swierstra.
- “Scala for the Impatient”⁷⁴, by Cay Horstmann.
- And finally, “Learn Functional Programming the Fast Way!”⁷⁵, by Alvin Alexander, also using Scala to teach concepts.

Cover photo by the author.

REFERENCES

- ¹ https://en.wikipedia.org/wiki/Categories_for_the_Working_Mathematician
- ² https://en.wikipedia.org/wiki/Saunders_Mac_Lane
- ³ https://www.reddit.com/r/mathmemes/comments/10tnkwu/based_definition/
- ⁴ https://en.wikipedia.org/wiki/Philip_Wadler
- ⁵ <https://deprogrammaticaipsum.com/search/?q=james+iry>
- ⁶ <https://james-iry.blogspot.com/2009/05/brief-incomplete-and-mostly-wrong.html>
- ⁷ https://en.wikipedia.org/wiki/Eugenio_Moggi
- ⁸ <https://doi.org/10.1109/LICS.1989.39155>
- ⁹ [https://doi.org/10.1016/0890-5401\(91\)90052-4](https://doi.org/10.1016/0890-5401(91)90052-4)
- ¹⁰ <https://sauropods.win/@futurebird/114392527567949088>
- ¹¹ <https://dl.acm.org/doi/abs/10.1145/91556.91592>
- ¹² https://link.springer.com/chapter/10.1007/3-540-15975-4_33
- ¹³ <https://dl.acm.org/doi/10.1145/143165.143169>
- ¹⁴ <https://dl.acm.org/doi/10.1145/158511.158524>
- ¹⁵ <https://sigplan.org/Awards/POPL/>
- ¹⁶ <https://deprogrammaticaipsum.com/in-praise-of-mathematics/>
- ¹⁷ <https://deprogrammaticaipsum.com/bertrand-russell/>
- ¹⁸ <https://deprogrammaticaipsum.com/douglas-hofstadter/>
- ¹⁹ https://en.wikipedia.org/wiki/Curry_%28programming_language%29
- ²⁰ [https://en.wikipedia.org/wiki/Church_\(programming_language\)](https://en.wikipedia.org/wiki/Church_(programming_language))
- ²¹ [https://en.wikipedia.org/wiki/G%C3%B6del_\(programming_language\)](https://en.wikipedia.org/wiki/G%C3%B6del_(programming_language))
- ²² <https://freesourcelibrary.com/the-legacy-of-russell-language/>
- ²³ <https://github.com/Frege/frege>
- ²⁴ <https://en.wikipedia.org/wiki/Rocq>
- ²⁵ https://en.wikipedia.org/wiki/Thierry_Coquand
- ²⁶ <https://dl.acm.org/doi/10.1145/2837614.2837647>
- ²⁷ https://en.wikipedia.org/wiki/Stephen_Cole_Kleene
- ²⁸ <https://deprogrammaticaipsum.com/joe-armstrong/>
- ²⁹ [https://en.wikipedia.org/wiki/Euler_\(programming_language\)](https://en.wikipedia.org/wiki/Euler_(programming_language))
- ³⁰ <https://deprogrammaticaipsum.com/issue/issue-065-pascal/>
- ³¹ <https://en.wikipedia.org/wiki/Napier88>
- ³² [https://en.wikipedia.org/wiki/Occam_\(programming_language\)](https://en.wikipedia.org/wiki/Occam_(programming_language))
- ³³ https://en.wikipedia.org/wiki/Z_notation
- ³⁴ https://en.wikipedia.org/wiki/Ernst_Zermelo
- ³⁵ <https://www.tandfonline.com/doi/abs/10.1080/0144534031000076237>
- ³⁶ https://link.springer.com/chapter/10.1007/978-3-642-40447-4_1
- ³⁷ https://amturing.acm.org/bib/backus_0703524.cfm
- ³⁸ <https://dl.acm.org/doi/10.1145/359576.359579>
- ³⁹ <https://www.cs.utexas.edu/~EWD/transcriptions/OtherDocs/Haskell.html>

- ⁴⁰ <https://people.cs.nott.ac.uk/pszgmh/faq.html>
- ⁴¹ <https://dl.acm.org/doi/10.1145/263699.263715>
- ⁴² <https://www.oreilly.com/pub/au/2440>
- ⁴³ <https://www.microsoft.com/en-us/research/publication/the-implementation-of-functional-programming-languages-2/>
- ⁴⁴ <https://www.goodreads.com/book/show/3791460-introduction-to-functional-programming>
- ⁴⁵ <https://dl.acm.org/doi/10.1145/99370.99404>
- ⁴⁶ <https://dl.acm.org/doi/10.1145/944705.944723>
- ⁴⁷ <http://portal.acm.org/citation.cfm?doid=143165.143169>
- ⁴⁸ <https://dl.acm.org/doi/10.1145/1238844.1238856>
- ⁴⁹ <https://www.cambridge.org/core/journals/journal-of-functional-programming/listing?q=wadler&searchWithinIds=49AD4731AAB0E94D8EF98BBB4EE56A7F&fts=yes>
- ⁵⁰ https://www.youtube.com/watch?v=gui_SE8rJUM
- ⁵¹ <https://dl.acm.org/doi/10.1145/274930.274933>
- ⁵² <https://dl.acm.org/doi/10.1145/286385.286387>
- ⁵³ https://link.springer.com/chapter/10.1007/978-3-642-60085-2_9
- ⁵⁴ <https://web.archive.org/web/19990209081452/http://www.cs.bell-labs.com/who/wadler/realworld/index.html>
- ⁵⁵ <https://homepages.inf.ed.ac.uk/wadler/realworld/>
- ⁵⁶ <https://pandoc.org/>
- ⁵⁷ <https://homepages.inf.ed.ac.uk/wadler/vita.pdf>
- ⁵⁸ <https://web.archive.org/web/20080901052313/http://book.realworldhaskell.org/>
- ⁵⁹ <https://web.archive.org/web/20081013110019/http://learnyouahaskell.com/>
- ⁶⁰ https://deprogrammaticaipsum.com/_why/
- ⁶¹ <https://haskellbook.com/>
- ⁶² <https://pragprog.com/titles/btlang/seven-languages-in-seven-weeks/>
- ⁶³ <https://simon.peytonjones.org/slpj-book-1987/>
- ⁶⁴ <https://dl.acm.org/doi/book/10.5555/113903>
- ⁶⁵ https://en.wikipedia.org/wiki/Structure_and_Interpretation_of_Computer_Programs
- ⁶⁶ <https://berksoft.ca/gol/>
- ⁶⁷ https://en.wikipedia.org/wiki/Practical_Common_Lisp
- ⁶⁸ <https://norvig.github.io/paip-lisp/#/>
- ⁶⁹ <https://www.cse.chalmers.se/research/group/logic/TypesSS05/Extra/geuvers.pdf>
- ⁷⁰ <https://www.fm2gp.com/>
- ⁷¹ <https://fsharpforfunandprofit.com/>
- ⁷² <https://pragprog.com/titles/swdddf/domain-modeling-made-functional/>
- ⁷³ <https://www.objc.io/books/functional-swift/>
- ⁷⁴ <https://www.informit.com/store/scala-for-the-impatient-9780138033651>
- ⁷⁵ <https://alvinalexander.gumroad.com/l/learnfp>