A Case for Kaeon FUSION

By Jesse Dinkin

PREFACE

For many months now, I've sealed myself away in my room, working on a passion project that consumed me to such a degree that it drove me out of university and away from what little a social life I had. A new programming language called Kaeon FUSION, created for the sole purpose of being the one programming language to rule them all, the perfect programming language. However, in the end I've come to the sobering realization that creating the perfect programming language is by its very nature a Sisyphean task, impossible regardless of one's technical aptitude. Programming languages are created for specific purposes, one cannot create a single programming language that encompasses the needs of all engineers. And now, after reviewing the results of my naïve efforts, and coming to terms with the truth of their futility, I believe that my hard work has paid off. I have succeeded in creating the perfect programming language.

Kaeon FUSION itself is described in the documentation within the Kaeon FUSION repository within the "Gallery-of-Kaeon" Github account. For brevity's sake and out of respect for your time, I will have the content of this essay focus primarily on how Kaeon FUSION came to be and why it is necessary, rather than on its technical attributes.

PART 1: ORIGINS

KML

Though the working world cries out for web and mobile developers, the bureaucratic halls of academia remain firmly rooted in console applications. Now, I personally consider computer science to be a liberal art, one that can only be fully appreciated with a solid grounding in theory, but it was clear that if I wanted to learn the tools of industry, I would have to do so outside the classroom. I bit the bullet and began teaching myself HTML through an online course.

After removing myself from my worldly concerns to concentrate, HTML was fairly easy to pick up. However, I wasn't satisfied. I came to understand the language, but I wasn't entirely sure how to apply my newfound skills to a real project. Thus, I turned my sights to a new endeavor. XML is ugly, and HTML suffers by extension, so I would create a newer, prettier XML. KML, or Kaeon Markup Language, became my new pet project.

Despite its shortcomings, KML was a significant step forward for me as a developer. Until that point, all of my projects were like bootlegged versions of products made by professionals. With KML, I finally had something one could make a solid case for against the industry standard. However, this is where the luster of KML begins to dim.

I could never come to conclusive decisions on many of KML's most important features, the majority of its functionality was hacked in as I thought of it, and every update required a new translator to be built. And, as is tradition with naïve language developers, the implementation was the specification. Therefore, a glitch in the translator was a glitch in the language. The most significant flaw was that it was difficult to make a case for using it over XML unless you were writing it by hand, which is rare with XML.

As I suffered through an internal crisis regarding KML, I was also thinking of designing other languages. It occurred to me that since KML was just a markup language, I could use its syntax as the base for an actual programming language. It was also around this time that I began learning about other markup languages and studying programming language design formally. Two things occurred to me during my studies: not all markup languages share XML's attribute system, and all languages essentially have one thing in common: that as they are parsed their tokens are sorted into a tree of strings.

ONE

It was upon these realizations that I had my critical breakthrough: to create a base language upon which any other language could be built, the key is to create a language solely for the purpose of defining a tree of strings. To serve as a template upon which various features could be implemented, I decided that this base language would not support any syntactic sugar. It would only allow for strings to be defined in literal character blocks, enforce strict rules regarding

whitespace and indentation, and neither have nor require any escape sequences, making it the perfect minimum viable product from which I could go in many directions. Unwilling to have it suffer from my previous mistakes, I defined it in a specification before implementing it. In fact, as a testament to how readable this new language was, the specification itself is written in the very language it describes. I christened my new language: ONE.

Using ONE as a template, I experimented with various backwards compatible mnemonics and preprocessors, eventually settling upon a set of such features I dubbed ONE+. However, no matter what sort of syntactic conventions I added, the semantics remained the same. Whether data is encoded in ONE or ONE+, the data itself is nothing more than a tree of strings. Thus, the functionality of a language built upon ONE is determined entirely by the content of the string tree.

TRINITY

Over the course of several months, I built several experimental programming languages on top of ONE, with my end goal being that any software development could eventually be done using ONE or ONE+ syntax. My first attempt was a language called Trinity. Instead of being a traditional programming language, Trinity allowed users to specify models that would cross compile custom languages built on ONE into other languages. Essentially, it was a customizable language translator for ONE. In theory this could have worked, but it was tedious to implement and even more tedious to use, so it was abandoned.

FUSION

It was only after months of wasting my time on Trinity that I had my second breakthrough. I figured that instead of designing an entire language based on ONE, I could create a system such that any ONE document could serve as code. In a similar manner to LISP or TCL, every element could be a command and objects returned by executing its children as commands would serve as arguments. On the recommendation of a friend, I decided to call this system: FUSION.

Again, similar to LISP, many dialects could be implemented upon this system. Due to the modular nature of the language, I could easily implement commands as they were needed, without worrying about how compatible they are with the rest of the language. Of course any implementation of the language would be interpreted, but commands could be added to generate compiled code at runtime, thus making the language infinitely versatile. I decided that since any functionality I want to implement would not jeopardize compatibility, that I could implement a single dialect that meets the needs of all developers, and finally, I would have a single language I could use that would allow me to branch out into all development domains. That is how Kaeon FUSION came to be.

PART 2: RELEVANCE

FRUSTRATION

The need to create a perfect language was born from my frustration as a software developer. The world of software is vast and ever expanding, as are the tools used for software creation. So fast is this dark energy of software, that no one developer could possibly hope to become well versed in more than a few aspects of development. Indeed, developers must specialize, and they must do so early. If a developer wishes to venture into a new domain, they must first allocate the time to carefully study an entirely new toolset, during which they must make connections with people who are familiar with the technology. Even after learning the basics, mastering the technology can take anywhere from years to decades. When one is already extremely comfortable in a given domain, the transition becomes even harder.

This frustration set in hard with me as I began attending hackathons in college. Until that point I'd been programming exclusively in Java, primarily on small games using the Swing toolkit. Now, Java may be a versatile tool, but it has its limits. At the time, web and mobile development were the hot topics among developers. Java is useless for web development, and for mobile development behaves very differently than it does for desktop development. As much as I wanted to branch out, it was fool's errand attempting to teach this old dog new tricks at a 36 hour event. Indeed I never did win a hackathon. My games, no matter how flashy they were, always lost out to some web app that made interesting use of the API of the week.

I figured that if I had a perfect language, I could easily branch into other development domains. Doing something I'd never touched upon before would be as simple as googling a new command.

Unfortunately, since I seemed to be alone amongst my peers in my opinion that this wasn't a pipe dream, it was clear this was something I'd have to pursue myself.

WHY KAEON FUSION IS A PERFECT LANGUAGE

A perfect language must meet the development needs of all engineers and it must do so elegantly. Normally, no one language can do this because, as stated in the preface, languages tend to be create for specific reasons, only to attempt to branch out later. However, what sets Kaeon FUSION apart is the fact that it is built on systems built for the express purpose of accommodating new features. Indeed the keys to Kaeon FUSION's ability to meet these criteria are the versatility and beauty of ONE and the modularity of FUSION.

Because any desired functionality can be implemented by adding a new command to the interpreter, any feature it lacks can be implemented without having to modify the existing language.

Additionally, adherence to the rules of the language do not matter until the interpreter reaches the element containing the code. Thus, ONE code following different formats can be injected into Kaeon FUSION code. In a similar manner to what was attempted with Trinity, such code can even be cross compiled to other languages.

CONCLUSION

I hold steadfast to my conviction that not only is Kaeon FUSION *a* perfect language, it is *the* perfect language. I cannot imagine any other way to create a language that accommodates the needs of all engineers and does so in an elegant manner.