Zen

Mitchell D. Goff

June 2, 2017

1 Project Overview

Zen is a new programming language for the web. It aims to be simple, readable, and concise, and it allows developers to write all their code in a single language. The Zen compiler, which I am submitting for consideration in Dartmouth's Kemeny Prize, transforms Zen expressions into JavaScript code, which can be executed by either an ES5-compatible web browser or a JavaScript runtime like Node.js. The compiler is currently written in a mixture of Python and Zen; however, I intend to port the remaining Python code to Zen once the language is capable of bootstrapping itself.

The claim that Zen might allow a web developer to write all their code in just one language is, in my mind, a rather astounding one. Web developers today use an astonishing array of languages and data formats, partly because these different languages are best suited to different things. For back-end work, a developer might use PHP, Python, Ruby, or JavaScript; for front-end work, they must use what web browsers will support (mostly HTML, CSS, and JavaScript); and for configuration files and project data, they might opt to use XML, JSON, or perhaps YAML.

Besides whatever problems these languages suffer from individually, they all share the subtle flaw of having been invented independently of each other. Real-life usage often forces languages to interact in complex and awkward ways, and these languages, not having been designed to work seamlessly together, suffer as a result. Front-end work tends to be especially problematic in this regard. When building a web app, a developer must write code that will load data from a remote API, use that data to generate complex HTML components, and display those HTML components to the user – which means dealing with HTML, CSS, and whatever data format the API uses (perhaps XML or JSON).

2 User Instructions

This section provides a brief overview of the Zen programming language. I realize that many of the judges for the Kemeny Prize will already be familiar with the concepts presented in this section, but for the sake of any non-technical readers, I'll do my best to explain the language as simply and clearly as possible, starting from the very beginning.

2.1 "hello, world"

"The only way to learn a new programming language is by writing programs in it." That's what Kernighan and Ritchie say, anyways, and the inventors of C would probably know; so let's jump right in with a "hello world" program. In Zen, this requires only one line of code, and looks something like this:

```
(print "Hello, world!")
```

This line of code is a full-fledged Zen program – in fact, if we place it in a file, we can use the Zen compiler to build and run it. To see how this works,

let's name our file `hello-world.zen`, just as an example. To run it, we open a shell and type in the following command:

```
>> zen run hello-world.zen
Hello, world!
```

So far, so good! Before we go any further, though, let's examine our "hello world" program a bit more closely. Readers who have spent some time programming in a language like LISP will already be familiar with the syntax that Zen uses, but if you're accustomed to writing in a language like C or Java or Python, and our program looks a little strange to you, not to worry! All we are doing here is calling a function – namely, the `print` function – and passing it the string "Hello, world!" as an argument.

Of course, calling a function is exactly the sort of thing you might do in C or Java or Python all the time – but in these languages, and a good many others, you put the function's name *before* the parentheses rather than *inside* them; so a "hello world" program might look something more like `print("Hello, world!")`. The Zen way of writing programs, where you put the function *inside* the parentheses, may seem strange at first to some readers, but as we'll see later on, this syntax lets us do some pretty nifty things that aren't possible in most other languages.

2.2 Functions

Passing around one argument at a time is fun and all, but why stop there? As a matter of fact, the `print` function will print out as many strings as we care to pass it. We've already done "hello world", so this time let's print something a little more zen:

```
>>> (print "words" "cannot" "open" "another's" "mind.")
words cannot open another's mind.
```

That's pretty Zen, I'd say! We gave `print` a whole bunch of strings, and it joined them all together. The arguments are just separated with a space – Zen ignores commas, so putting them between your arguments won't do anything.

Now, let's try making a function of our own. In Zen, the way to define a function is like this:

```
(def print-twice (x)
    (print x)
    (print x))

>>> (print-twice "hungry hungry hippos")
hungry hungry hippos
hungry hungry hippos
```

Something funny is going on with all these parentheses: we're calling `def` just as if it were a function! It isn't, though – it's one of Zen's special primitives.

These primitives can do some very fancy things, but `def` is pretty straightforward; we give it the function's name, then its arguments, then its code. (Whitespace doesn't have any special meaning in Zen, so you can indent your code however you prefer.)

2.3 Operators

Now that we've got functions out of the way, let's have some fun with numbers!

```
>>> 2 + 3
5
>>> 5 * 5 * 5
125
>>> (7 - 4) / 1.2
2.5
```

This is pretty self-explanatory. Zen uses the standard arithmetic operators and obeys the usual precedence rules, and we can use parentheses to group terms together. Operators are evaluated before function calls, so although it might look in the third example as though we're trying to call the `7` (with `-` and `4` as arguments), we actually evaluate the operator first (getting us `3`) and then call the `3` with no arguments. In Zen, all values return themselves when called with no arguments, so `((("a rose")))` has exactly the same value as `"a rose"` no matter how parentheses we surround it with.

One other little pitfall to watch out for is that Zen allows you to use a few special symbols (`+, -, *, /, _,` and `?`) in your variable names, so you can call your functions things like `print-twice` or `*alert*` or `empty?`. This is very nice, but we have to be careful to put spaces around operators that use these symbols – `x+y` is a valid name in Zen, so if we want to add `x` and `y` together, we must write `x + y` or the compiler will get confused and complain that it doesn't know anything about the variable `x+y`.

```
(def double (x)
    (x * 2))

>>> (double 2)
4
```

2.4 Pattern Matching

A lot of times, when we're assigning values to things in a programming language

3 Technical Description

This section will briefly describe the internals of the Zen compiler. Most of the modules are well documented with comments that explain the individual classes

and functions, so this section will focus on just the most important modules and how they fit together.

Life for a Zen program begins with the Linker (`zen/compile/js/linker.py`). The Linker's job is to compile a Zen module, along with the Zen Prelude and all of the module's imports, into a single executable JavaScript file. It does this in four stages: Parsing, AST transforms, compiling, and rendering.

3.1 Parsing

Parsers are fairly well-studied algorithms, and since the Zen parser is not especially complex, I will comment only briefly. Once the Linker has located and loaded the source code of the module being compiled, the code is passed as a string to the Lexer (`zen/parse/lex.py`), which transforms it into a list of tokens. The Parser (`zen/parse/parse.py`) then transforms this list of tokens into a list of abstract syntax tree (AST) nodes. The classes for these nodes can be found at `zen/ast.py`.

3.2 AST Transforms

Before a module's code (now in Zen AST form) is compiled into JavaScript, we need to modify the AST a little bit. We do this by passing the AST through several mapping functions, each of which performs a particular transform on all of the AST nodes and their children by calling itself recursively. At the moment, the Zen compiler uses only two AST transforms, `resolveDecorators` and `resolveFixity`, which can be found (along with their documentation) at `zen/transforms/decorators.py` and `zen/transforms/infix.py`, respectively.

3.3 Compiling

The next task is to transform a module's code, still in Zen AST form, into a JavaScript AST. JavaScript's AST is a bit more complex than Zen's ...

3.4 Rendering

Finally, the linker needs to render everything into a single JavaScript file. To do this, it loops through all the modules it has compiled so far, in the same order it compiled them, and calls their `.write()` methods. The module's `.write()` method calls the `.write()` method of each of its top-level JavaScript AST nodes, and they in turn call the `.write()` methods of each of their component nodes. Using `.write()`, the entire AST structure recursively renders itself into a single string of JavaScript code, which is passed back down the call stack to the linker. Once the linker has rendered every module, it joins them together and writes them to whatever output file the user has requested.

4 Evaluation