(This slide intentionally left blank)

So you want to write an interpreter?

Alex Gaynor, PyCon 2013

Friday, March 15, 13

Hi everyone. Thanks for coming. This talk is going to be about how to write your own interpreter. This talk is going to go pretty quickly, there's a lot of material to cover, but all the code is going to be online, so don't worry about copying every word down.

rdio.com

First, thanks to my employer, rdio.com, we’re a streaming internet music service, it’s really awesome, we’re hiring, you should come talk to me.

Who am I?

- DSF/PSF
- I work on PyPy, CPython, Django
- I created Topaz

What is an interpreter?

- It is a program. Just a bunch of code like any other.
- Its input is a program in some language.
- Its output is the result of running that prgoram.

- Not magic!

What is an interpreter? A lot of people consider interpreters and compilers to be some sort of black magic, a dark art, that you have to be indoctrinated into or something. They’re not (or maybe this is the indoctrination). They’re just programs.

Why do you want to write an interpreter?

- Create your own language?
- Implement an existing language?
- Have fun!

Why would you want to write one? Well to create a new language, to reimplement an existing language (I like doing this one a lot). Most of all, it can be a ton of fun, there’s lots of really cool mini-puzzles that go into this.

The pieces

- Lexer
- Parser
- AST
- Bytecode-compiler
- Bytecode interpreter
- Runtime

So a VM has basically 6 parts, and we’re going to go through each of them.

On your marks...

Get set...

Go!

Basically for the rest of this talk we're going to build a VM for a tiny subset of Javascript. From the ground up, it's going to be really awesome.

Our language

```
a = 3;
if (a >= 2) {
    print "a is big!";
}
else {
    print a;
}
```

Our language contains just a few features, but it’s really enough to do almost anything we want. The features we have are: numbers (floats) and strings, a few binary operators for comparisons and arithmetic, an if/else statement, and a print statement (that’s not part of javascript, I made it up).

```
pip install rply
```

RPly is a toolkit we're going to use to help us construct our parser and lexer.

Lexers!

So the very first step in writing an interpreter is writing a lexer.

What is a lexer?

A lexer basically takes a program, and turns it into a list of tokens. What is a token a token is a symbol in your program, let’s take a look at an example.

```
a = 3;
if (a >= 2) {
    print "a is big!";
}
else {
    print a;
}
```

```
[
    Token("NAME", "a"),
    Token("EQUAL", "="),
    Token("SEMICOLON", ";"),
    Token("IF", "if"),
    Token("LPAREN", "("),
    Token("NAME", "a"),
    Token("GREATER_EQUAL", ">="),
    Token("NUMBER", "2"),
    Token("LBRACE", "{"),
    Token("PRINT", "print"),
    Token("STRING", "a is big!"),
    Token("SEMICOLON", ";"),
    Token("RBRACE", "}"),
    Token("ELSE", "else"),
    Token("LBRACE", "{"),
    Token("PRINT", "print"),
    Token("NAME", "a"),
    Token("SEMICOLON", ";"),
    Token("RBRACE", "}"),
]
```

So that program, becomes that list of tokens. A token is basically two things, 1) it has a name, and b) it has the text it matches, so an "LBRACE" token always has the same text, but a NAME or NUMBER has different text. Also, notice that there's no whitespace, it's not significant so we just ignore it.

WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.
OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!
BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!
IT'S HOPELESS!
EVERYBODY STAND BACK.
I KNOW REGULAR EXPRESSIONS.
PERL!
TAP TAP

Friday, March 15, 13

How do we implement lexers? Regular expressions! Each token type is just a specific regular expression. Really simple.

```
from rply import LexerGenerator
lg = LexerGenerator()
lg.ignore(r"\s+")
lg.add("IF", r"if")
lg.add("ELSE", r"else")
lg.add("PRINT", r"print")
lg.add("LPAREN", r"\(")
lg.add("RPAREN", r"\)")
lg.add("LBRACE", r"\{")
lg.add("RBRACE", r"\}")
lg.add("EQUAL", r"=")
lg.add("GREATER_EQUAL", r">=")
lg.add("SEMICOLON", r";")
lg.add("NUMBER", r"\d+")
lg.add("NAME", r"[a-zA-z_][a-zA-Z0-9_]*")
lexer = lg.build()
```

So this is basically how lexers in RPly work. You create a generator, you add the rules you want, with a name and a regexp, and then you create

```
>>> from minijs.lexer import lexer
>>> stream = lexer.lex("my_var = 23")
>>> stream
<rply.lexer.LexerStream object at 0x10bfb2090>
>>> stream.next()
Token('NAME', 'my_var')
>>> stream.next()
Token('EQUAL', '=')
>>> stream.next()
Token('NUMBER', '23')
>>> stream.next()
>>>
```

As you can see, we get a LexerStream, which yields us one token at a time.

Parsers

So in general, each stage of our interpreter is going to take the output of the previous stage and do something with it. So our first stage took the source code and gave us a list of tokens. Now we need to take a list of tokens and do something with it.

```
[
    Token("NAME", "a"),
    Token("EQUAL", "="),
    Token("SEMICOLON", ";"),
    Token("IF", "if"),
    Token("LPAREN", "("),
    Token("NAME", "a"),
    Token("GREATER_EQUAL", ">="),
    Token("NUMBER", "2"),
    Token("LBRACE", "{"),
    Token("PRINT", "print"),
    Token("STRING", "a is big!"),
    Token("SEMICOLON", ";"),
    Token("RBRACE", "}"),
    Token("ELSE", "else"),
    Token("LBRACE", "{"),
    Token("PRINT", "print"),
    Token("NAME", "a"),
    Token("SEMICOLON", ";"),
    Token("RBRACE", "}"),
]
```

$\longrightarrow$

```
Block([
    Assignment("a", Number(3)),
    If(
        Comparison(">=", Name("a"), Number(2)),
        Block([
            Print(String("a is big!")),
        ]),
        Block([
            Print(Name("a")),
        ])
    )
])
```

So basically we take our list of tokens, and turn it into a tree which represents what's going on. A Block is just a group of operations. Assignment is assignment. Name is a variable load. etc. All the nodes are pretty simple. This is called an abstract syntax tree.

```
class Node(object):
    def __eq__(self, other):
        if not isinstance(other, Node):
            return NotImplemented
        return (type(self) is type(other) and
            self.__dict__ == other.__dict__)
    def __ne__(self, other):
        return not (self == other)
class Block(Node):
    def __init__(self, statements):
        self.statements = statements
class Statement(Node):
    def __init__(self, expr):
        self.expr = expr
class Number(Node):
    def __init__(self, value):
        self.value = value
```

Friday, March 15, 13

So this is our initial set of AST classes, sorry about the very un-pep8 whitespace. Basically each one of these is a wrapper for some other stuff. We'll see why we have Statement later once we get to compilation. Now we need the parser that actually parsers.

```python
from rply import ParserGenerator
from minijs import ast
pg = ParserGenerator(["NUMBER", "SEMICOLON"], cache_id="minijs")
parser = pg.build()

@pg.production("main : statements")
def main(s):
    return s[0]
```

So basically the parser gets a set of rules, and an action to perform when there's a match for each rule. I consider this to be by far the most confusing and hardest part of writing an interpreter, so if it's confusing... welcome to the club.

```
@pg.production("statements : statements statement")
def statements(s):
    return ast.Block(s[0].getastlist() + [s[1]])

@pg.production("statements : statement")
def statements_statement(s):
    return ast.Block([s[0]])

@pg.production("statement : expr SEMICOLON")
def statement_expr(s):
    return ast.Statement(s[0])

@pg.production("expr : NUMBER")
def expr_number(s):
    return ast.Number(float(s[0].getstr()))
```

If you look you can see how this builds up. A group of statements is either one statement, or a list of statements. A statement is an expression with a semicolon, an expression is a number. And of course we'd want to expand this. For example addition is also an expression. There's tons of things we don't have here.

```python
@pg.production("expr : expr PLUS expr")
def expr_binop(s):
    return ast.BinOp(s[1].getstr(), s[0], s[2])
```

Friday, March 15, 13

So this rule is a fun one. expr + expr. What can this parse? It can handle 1 + 1 + 1, or anything else like that because the left and right hand sides *are* exprs, but the result is also an expr.

Compiler

So the next step is taking the AST we just produced, and turning it into something for an interpreter.

Bytecode

- A sequence of instructions for the machine
- Sometimes have arguments.
- Stack based.

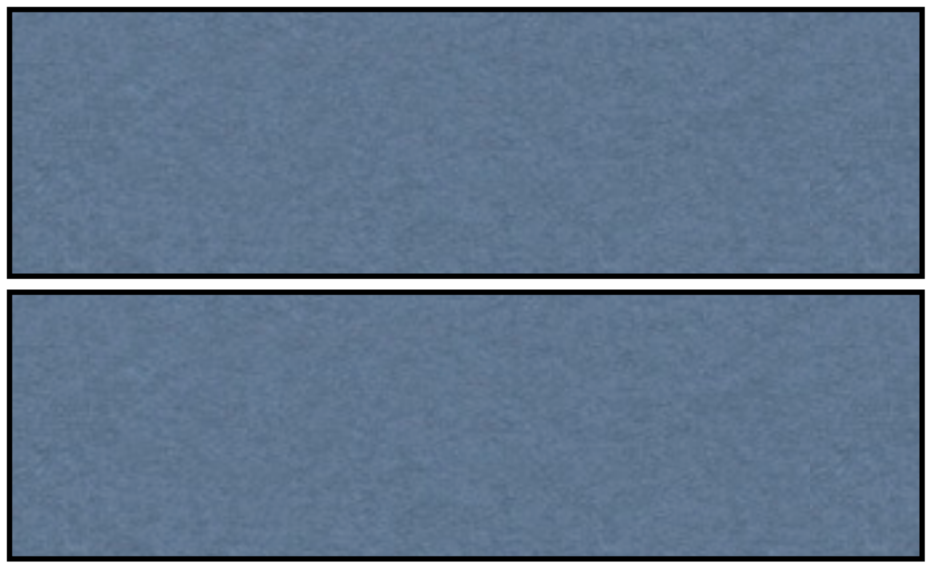
```
1 + 21;
```

```
LOAD_CONST 0
LOAD_CONST 1
BINARY_ADD
POP_TOP
```

So how does this work, we’ve got a few instructions here. LOAD_CONST, BINARY_ADD, POP_TOP. How does this work? We see the number 1. That becomes a LOAD_CONST, what’s that 0? The 0 means refer the the 0th constant, and we’ll keep an array of them somewhere else. Then we load up const number 1 (the second const). We now have 2 items on our stack. Then we add them togehter, that pops 1 and 21 off the stack, adds them, and pushes the result.

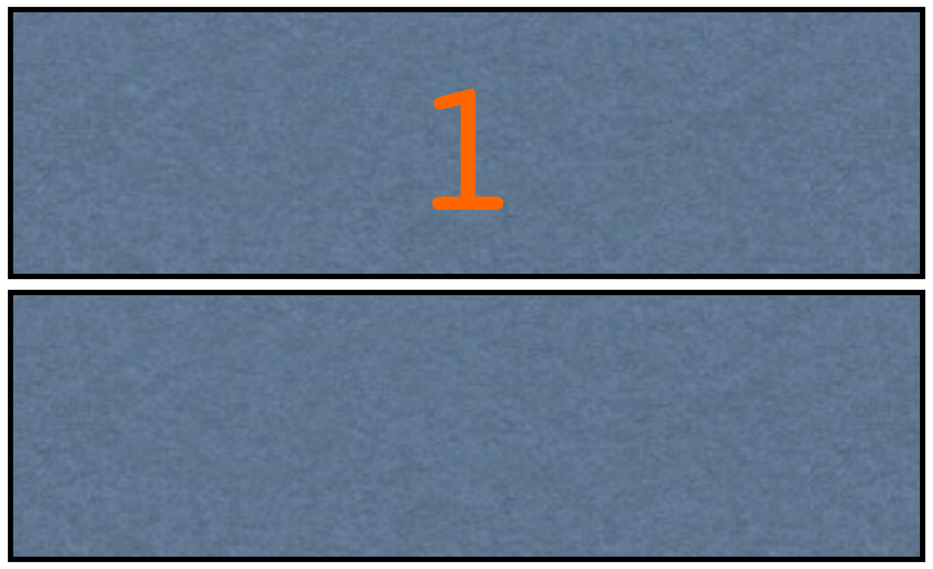
```
1 + 21;
```

```
LOAD_CONST 0
LOAD_CONST 1
BINARY_ADD
POP_TOP
```



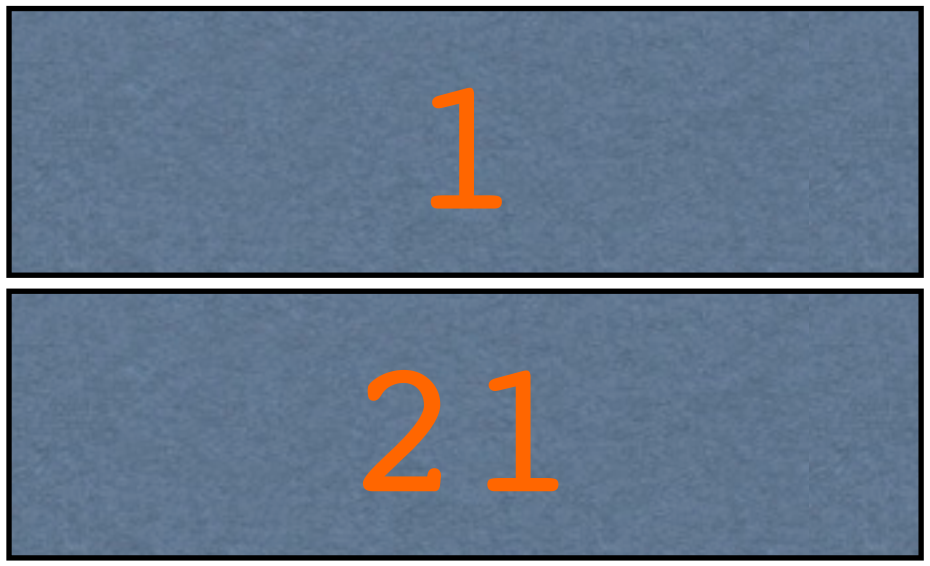
```
1 + 21;
```

```
➔LOAD_CONST 0
 LOAD_CONST 1
 BINARY_ADD
 POP_TOP
```



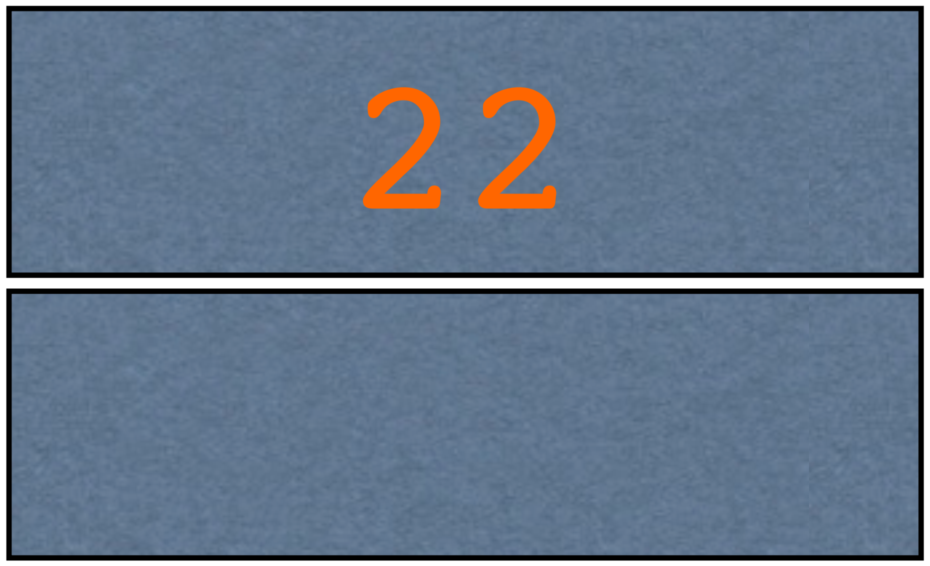
```
1 + 21;
```

```
  LOAD_CONST 0
→ LOAD_CONST 1
  BINARY_ADD
  POP_TOP
```



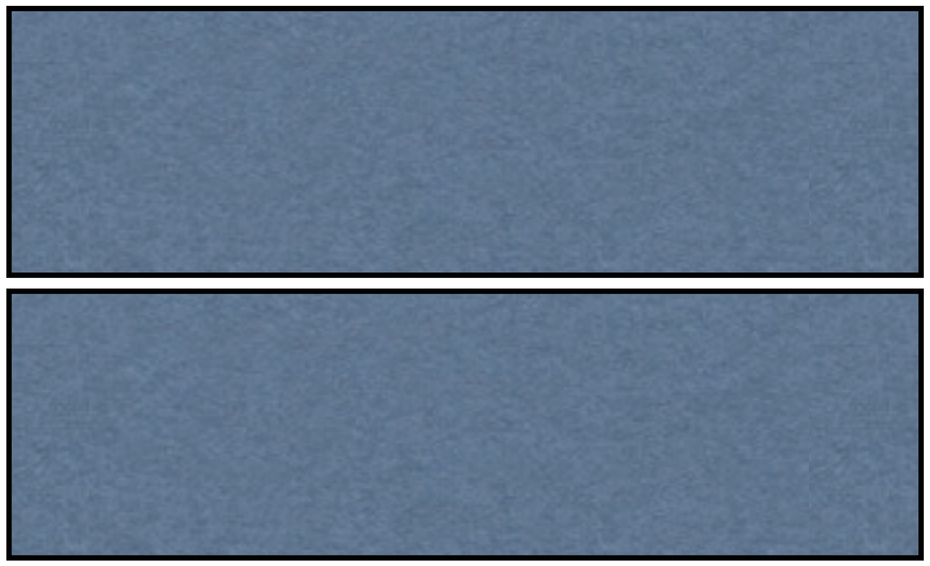
```
1 + 21;
```

```
 LOAD_CONST 0
 LOAD_CONST 1
→BINARY_ADD
 POP_TOP
```



```
1 + 21;
```

```
LOAD_CONST 0
LOAD_CONST 1
BINARY_ADD
POP_TOP
```



```
class Statement(Node):
    def compile(self, ctx):
        self.expr.compile(cxt)
        ctx.emit(POP_TOP)

class Number(Node):
    def compile(self, ctx):
        ctx.emit(LOAD_CONST, ctx.new_const(JSNumber(self.value)))
```

```
class BinOp(Node):
    def compile(self, ctx):
        self.left.compile(ctx)
        self.right.compile(ctx)
        opname = {
            "+": "BINARY_ADD",
        }
        ctx.emit(opname[self.op])
```

Object Model

```
class JSObject(object):
    pass

class JSNumber(JSObject):
    def __init__(self, value):
        self.value = value
```

```
class JSNumber(JSObject):
    def add(self, other):
        assert isinstance(other, JSNumber)
        return JSNumber(self.value + other.value)
```

Interpreter
(aka the Holy Grail)

```
class Interpreter(object):
    def interpret(self):
        pc = 0
        while pc < len(self.bytecode):
            opcode = self.bytecode[pc]
            opname = OPCODE_TO_NAME[opcode]
            pc = getattr(self, opname)(pc)
```

```python
class Interpreter(object):
    def LOAD_CONST(self, pc):
        arg = ord(self.bytecode[pc + 1])
        self.push(self.consts[arg])
        return pc + 2
```

```python
class Interpreter(object):
    def BINARY_ADD(self, pc):
        right = self.pop()
        left = self.pop()
        self.push(left.add(right))
        return pc + 1
```

Left as an excersise

- (PS: Come and find me or other PyPy/ interpreter geeks to chat about these)
- if/while
- More types (strings!)
- More everything

Useful references

- Topaz: http://github.com/topazproject/topaz
- Kermit Example Interpreter: https://bitbucket.org/pypy/example-interpreter/
- #pypy #topaz on Freenode

Recap

- Lexer
- Parser
- AST
- Compiler
- Object Model
- Interpreter

Go forth and have fun

Thanks, have an amazing PyCon!

http://speakerdeck.com/u/alex