

# Parsing

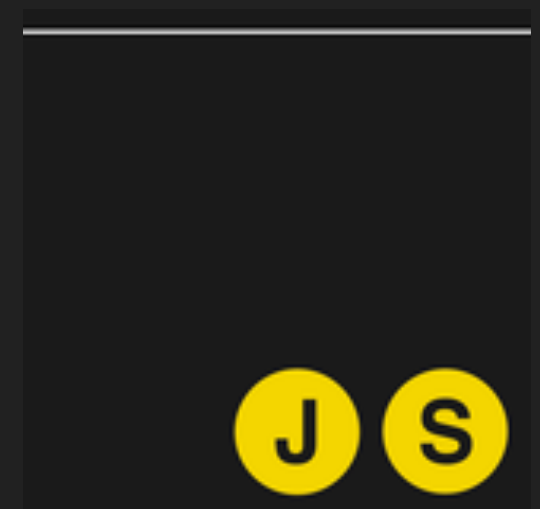
# Parsers

Jenna Zeigen  
JSConf Hawaii  
2/5/2020

# Senior Frontend Engineer at Slack

## Organizer of EmpireJS

## Organizer of BrooklynJS



@zeigenvector

jenna.is/at-jsconfhi

parsing

parsers!

# 1. abcs of language

1. abcs of language

2. hmm, actually,  
let's just step  
through a (small)  
parser

# the abcs of Language

# the abcs of language

## "language" is a structured system of communication

First you're up and you're down And  
then between Oh I really want to know  
What do you mean? Ooh





# the abcs of language

"natural language" is a  
naturally evolved system  
that humans use to  
communicate with each other

You speak  
And I know just what  
You're what you're sayin'  
♪ ♪ ♪



# the abcs of language

"formal languages" have an alphabet and words, which can be combined correctly based on specific rules

I got new rules, I count 'em  
I got new rules, I count 'em  
♪ ♪ ♪



# grammar school

a language's grammar is  
the set of rules for  
that language

Stop! Grammar time!



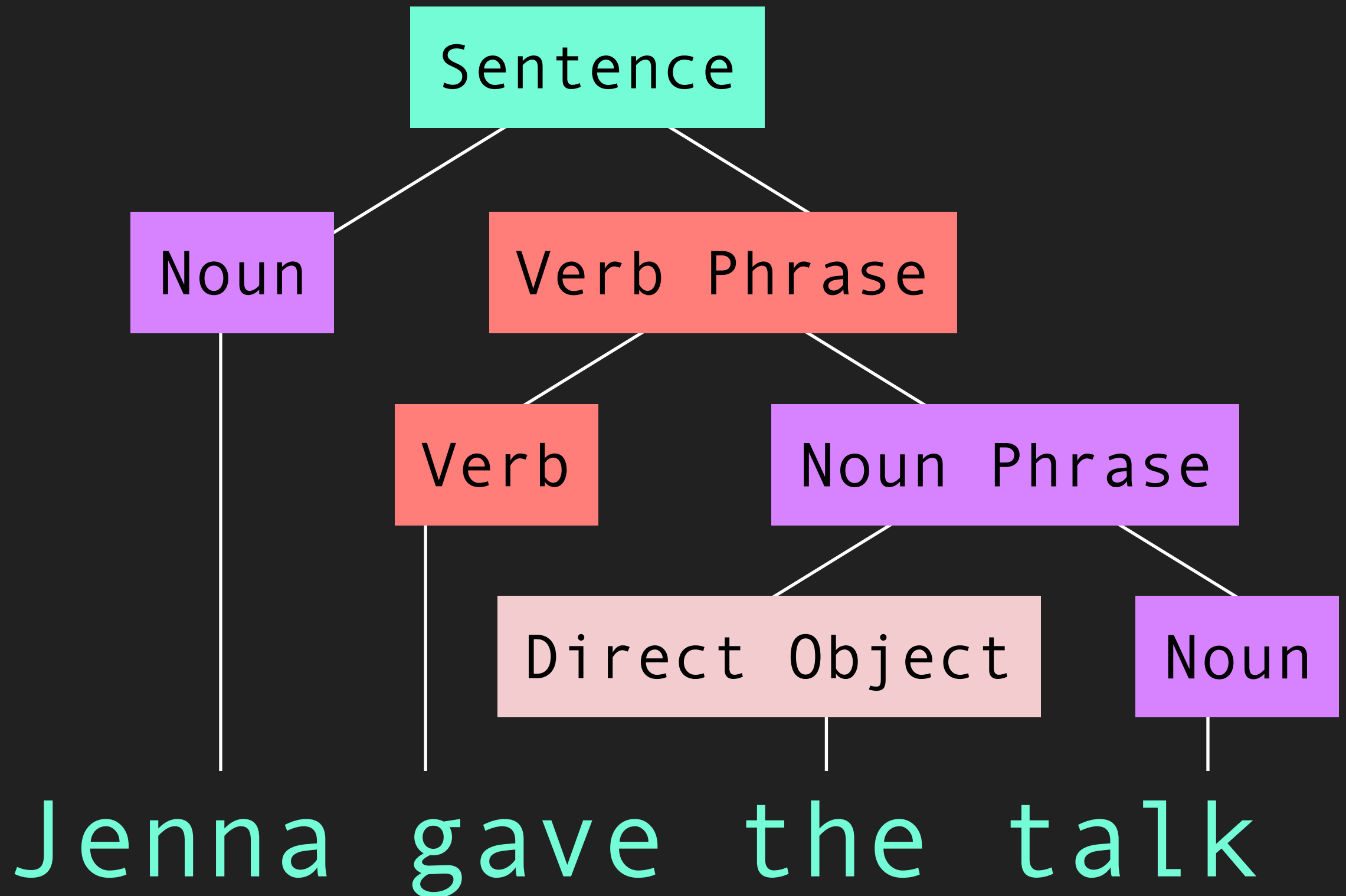
# grammar school

"formal grammars" put  
these rules in terms of  
replacement

To the left, to the left  
To the left, to the left (Mmm)  
To the left, to the left  
Non-terminals in the spot to the left  
To the left, to the left  
The grammar tells us for what symbols  
They are replaceable  
♪ ♪ ♪



# grammar school



# grammar school

Sentence = Noun + Verb  
Phrase

Verb Phrase = Verb +  
Noun Phrase

Noun Phrase = Direct  
Object + Noun



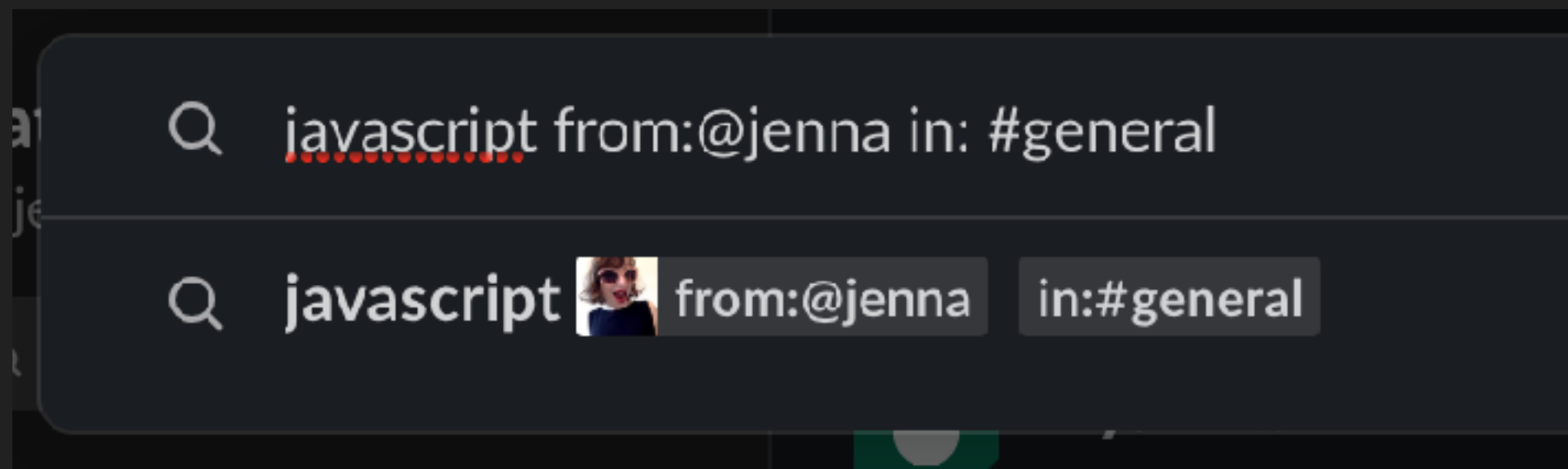
# grammar school

Programming language  
grammars are defined in  
their spec

```
thank u, spec  
thank u, spec  
thank u, spec  
I'm so very grateful for my spec  
♪ ♪ ♪
```



# syntax city





# syntax city

```
javascript "front end"  
in:#random in: #general  
from:@jenna
```

# syntax city

```
javascript "front end" in:#random  
in: #general from:@jenna
```

Query  $\rightarrow$  Term

Query  $\rightarrow$  Term Query

Query  $\rightarrow$  Filter

Query  $\rightarrow$  Filter Query

# syntax city

javascript	"front end"
------------	-------------

in: #random	in: #general
-------------	--------------

from: @jenna
--------------

ok, now  
parsers

# moving parse

the process of analyzing  
language against the  
rules of its grammar

I got my rules up,  
And a bit of language  
Is its syntax okay?  
Yeah we're parsing in the USA  
♪ ♪ ♪



# moving parse

a function that takes raw input and returns meaningful data created from the input, or an error

All the beautiful inputs  
Are very, very meaningful  
You know, space is my favorite delimiter  
I felt so symbolic yesterday

♪ ♪ ♪



# moving parse

parsers usually have  
two parts: the lexer  
and the parser

lexer and parser  
making us a tree  
P-A-R-S-I-N-G



@zeigenvector

jenna.is/at-jsconfhi

# lex go

the lexer takes the text  
and breaks it down into  
meaningful units, called  
"tokens"

```
Reading through this code  
I've been asked to invoke  
Got a lexer out here first  
Made a nice short token
```

♪ ♪ ♪





# lex go

first, the "scanner" goes through and breaks the string of characters into the proper chunks, or "lexemes"

I was born to lex (Yes)  
According to the spec  
What amazing tech,  
Having this effect (Woo)  
And soon the parser will turn  
These strings into objects (Money)  
♪ ♪ ♪



# lex go

coding time



# lex go

```
const lexemes =  
'Jenna gave the  
talk'.split('');
```

# lex go

const

lexemes

=

"Jenna gave the talk"

.

split

(

' '

)

;

# lex go

then, the "evaluator"  
combines the lexeme's type  
with its value to create  
the "token"

I then begin to encounter with my parse,  
To split the text apart  
Break it down into sections  
Tokens from the lexemes

♪ ♪ ♪



# lex go

coding time



# lex go

const

lexemes

=

"Jenna gave the talk"

.

split

(

' '

)

;

# lex go

Keyword

Identifier

Punctuator

String

Punctuator

Identifier

Punctuator

String

Punctuator

Punctuator



# lex go

```
[  
  { "type": "Keyword", "value": "const" },  
  { "type": "Identifier", "value": "lexemes" },  
  { "type": "Punctuator", "value": "=" },  
  { "type": "String", "value": "'Jenna gave a talk'" },  
  { "type": "Punctuator", "value": "." },  
  { "type": "Identifier", "value": "split" },  
  { "type": "Punctuator", "value": "(" },  
  { "type": "String", "value": "' '" },  
  { "type": "Punctuator", "value": ")" },  
  { "type": "Punctuator", "value": ";" }  
]
```

weird lex but ok



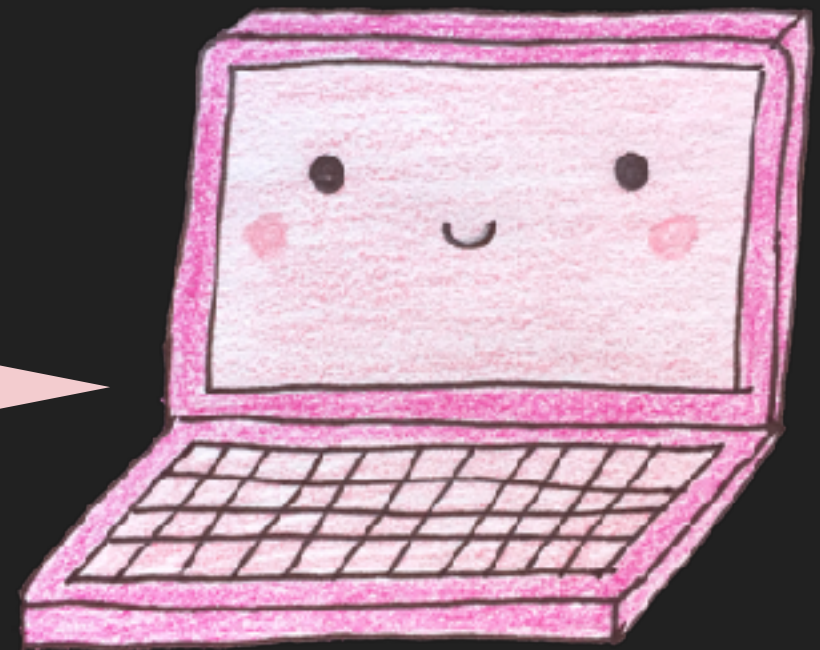
@zeigenvector

jenna.is/at-jsconfhi

# parse for the course

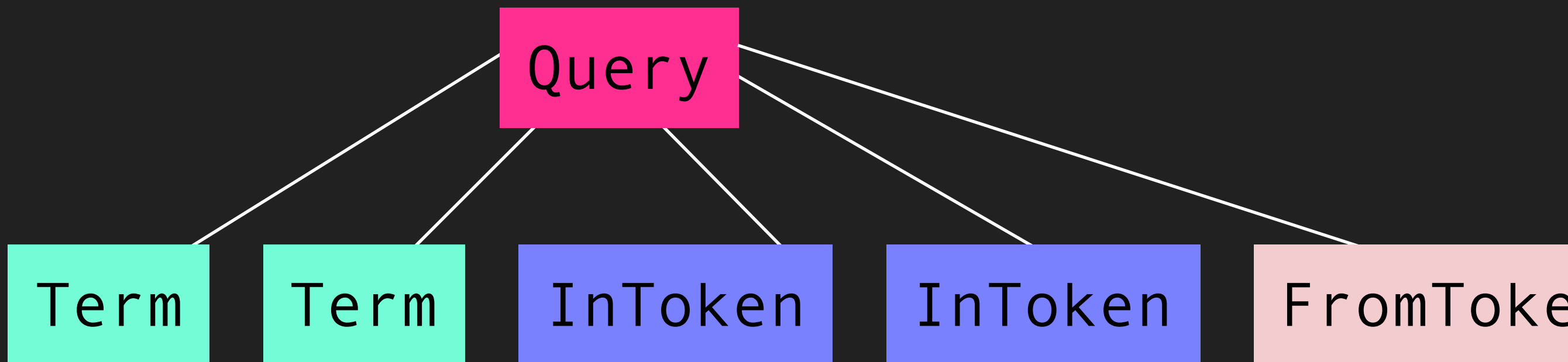
the parser will check that  
the syntax is correct while  
creating a structural  
representation

Every single word  
Is perfect as it can be  
And I put it in a tree



# parse for the course

javascript "front end" in:#random  
in: #general from:@jenna



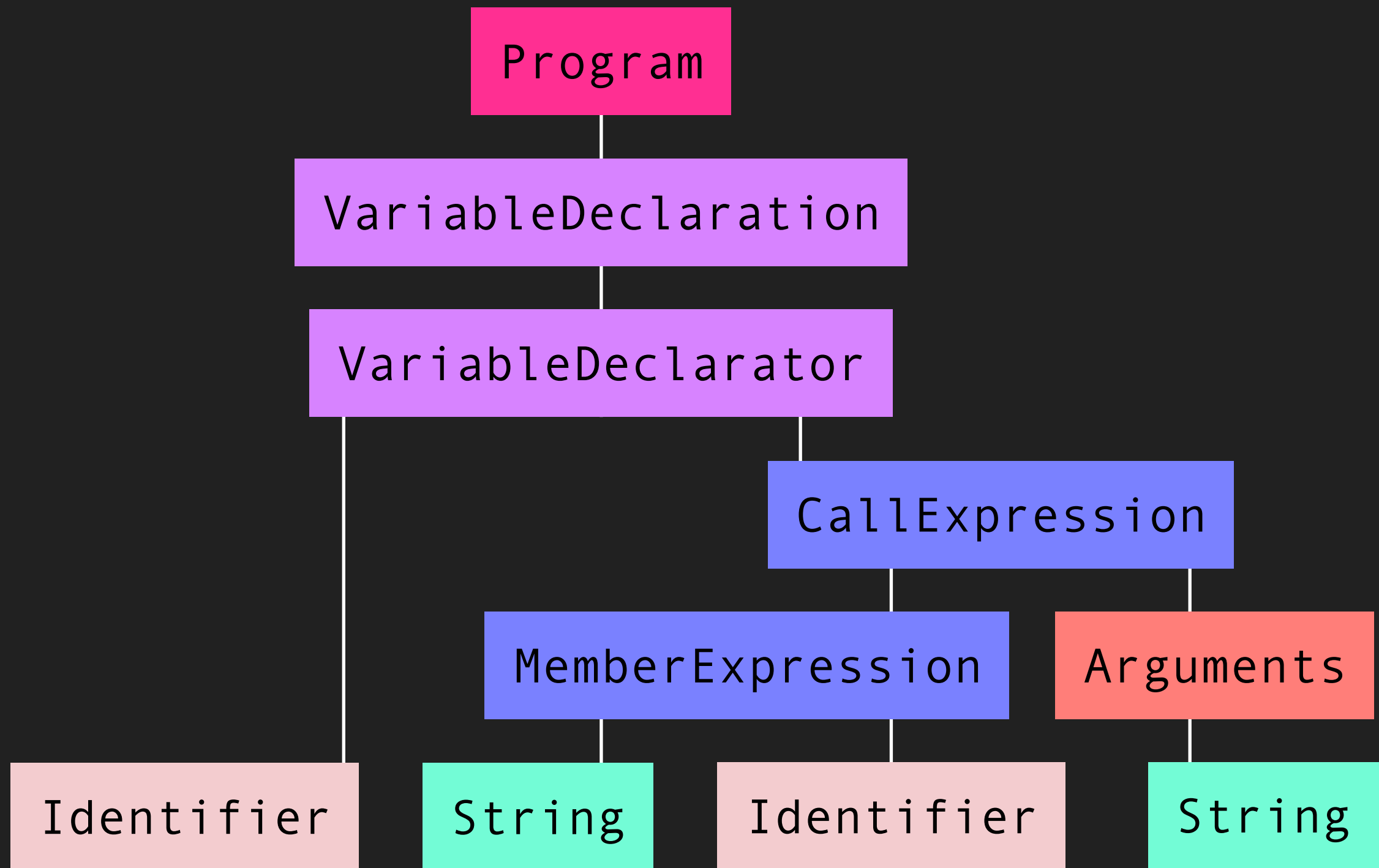
# parse for the course

I know who I want  
To read my code  
(It's you!)

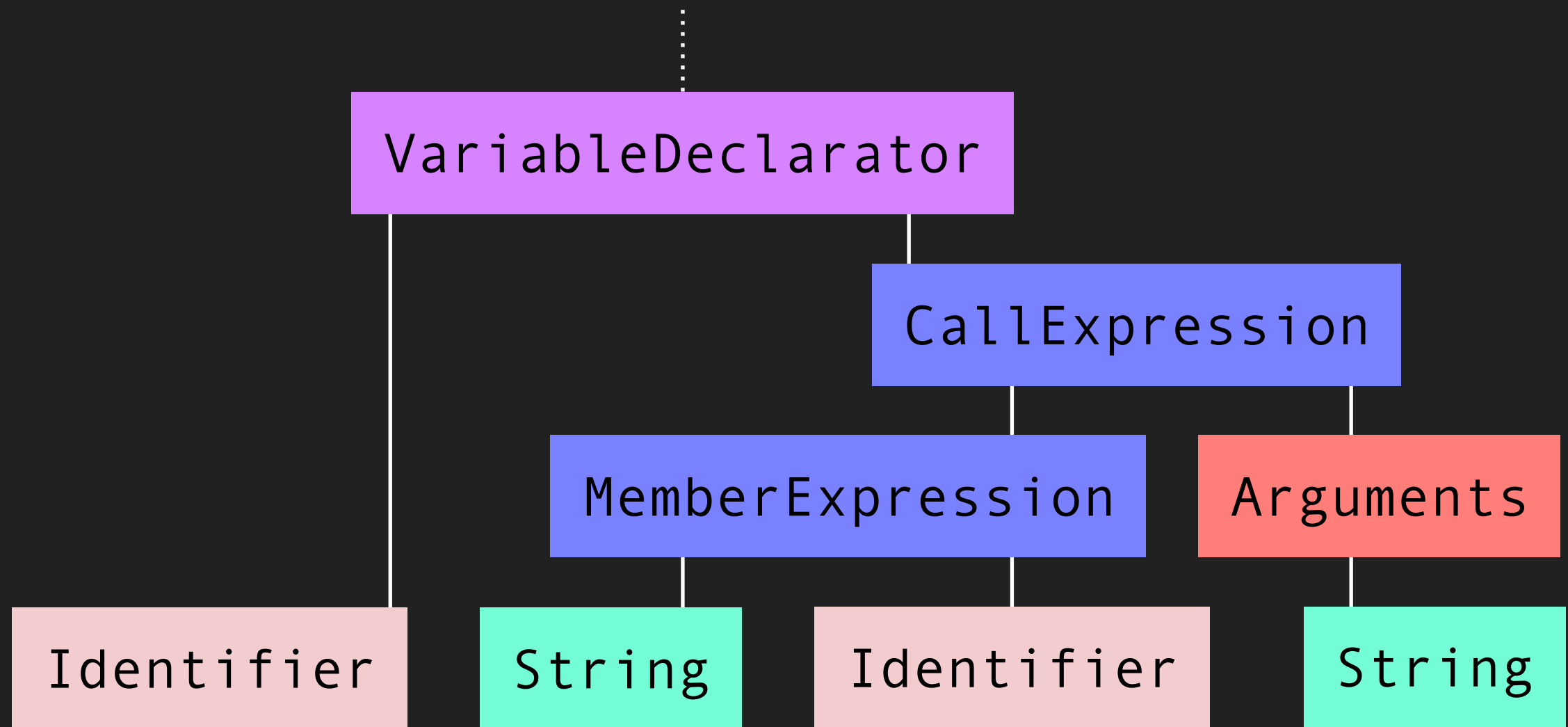
♪ ♪ ♪



# parse for the course



# parse for the course

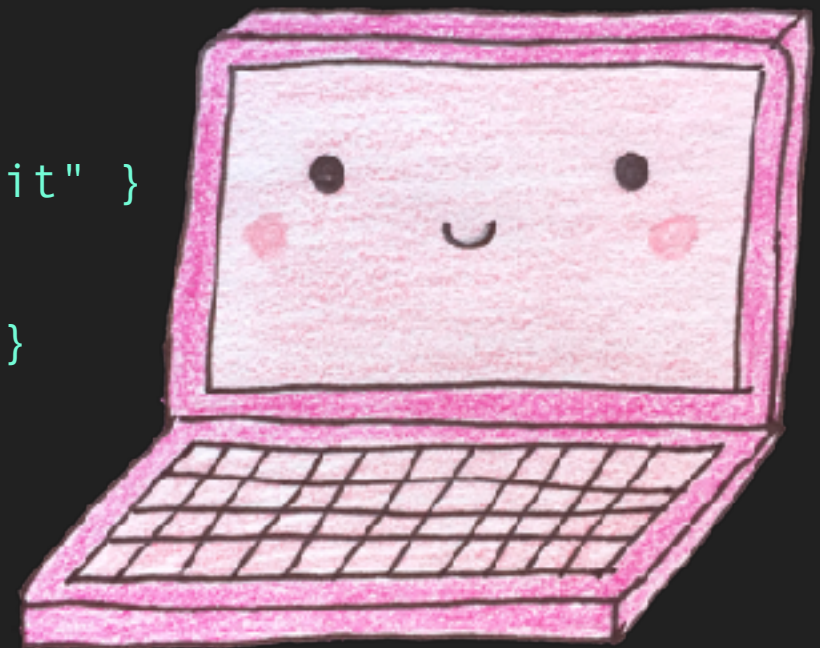


```
const lexemes = 'Jenna gave  
the talk'.split('');
```

# parse for the course

```
{
  "type": "Program",
  "body": [
    {
      "type": "VariableDeclaration",
      "declarations": [
        {
          "type": "VariableDeclarator",
          "id": { "type": "Identifier", "name": "lexemes" },
          "init": {
            "type": "CallExpression",
            "callee": {
              "type": "MemberExpression",
              "computed": false,
              "object": {
                "type": "Literal",
                "value": "Jenna gave the talk",
                "raw": "'Jenna gave a talk'"
              },
              "property": { "type": "Identifier", "name": "split" }
            },
            "arguments": [
              { "type": "Literal", "value": " ", "raw": "' '" }
            ]
          }
        }
      ],
      "kind": "const"
    }
  ]
}
```

Computers can have a  
little JavaScript,  
as a tree



@zeigenvector

jenna.is/at-jsconfhi



# syntax city

javascript

"front end"

in: #random

in: #general

from: @jenna



# syntax city

```
javascript "front end" in:#random  
in: #general from:@jenna
```

Query  $\rightarrow$  Term

Query  $\rightarrow$  Term Query

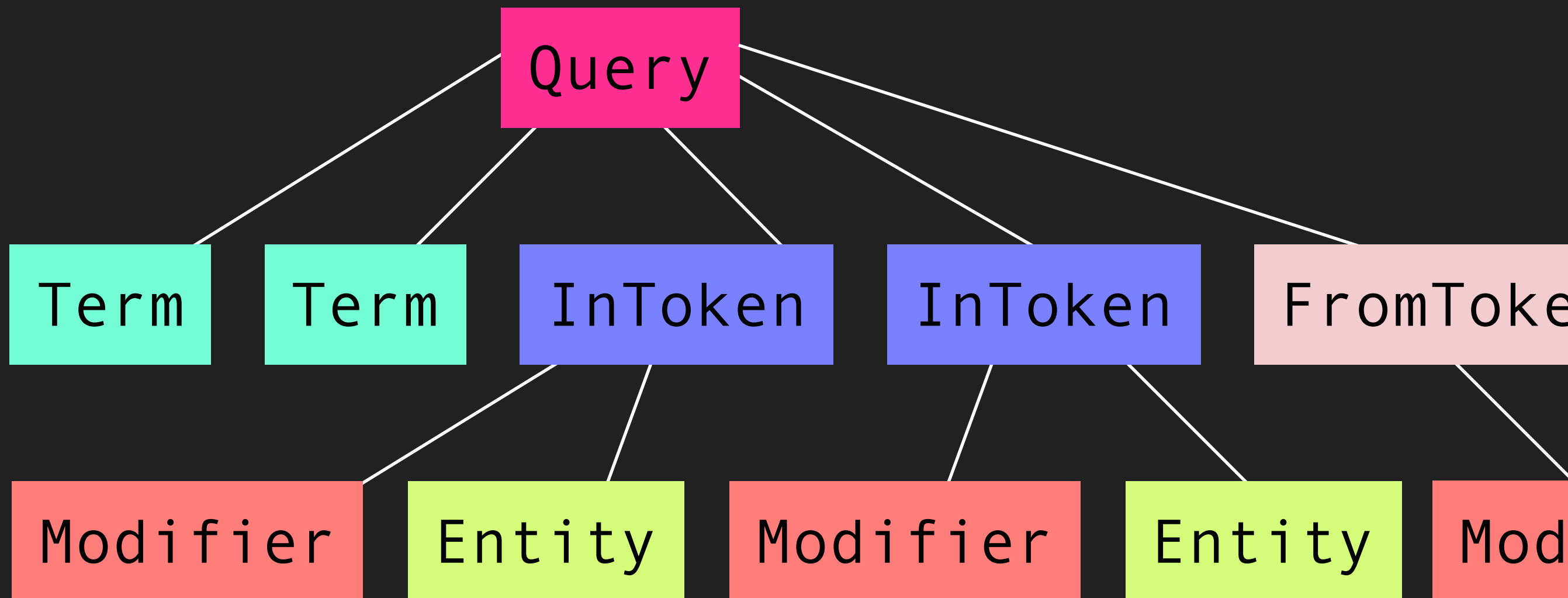
Query  $\rightarrow$  Filter

Query  $\rightarrow$  Filter Query

Filter  $\rightarrow$  Modifier Entity

# syntax city

javascript "front end" in:#random  
in: #general from:@jenna



# parse for the course

Read my co-odeeee



the more  
complicated  
stuff...

# advanced grammar school

/in: ?([ ^ ]+)|from: ?  
([ ^ ]+)'|"([ ^"]+)"|  
\'([ ^\' ]+)\'|([ ^ ]+)' /

# advanced grammar school

A "regular grammar" is  
one where all the  
production rules are  
one of the following:

$$A \rightarrow a$$

$$A \rightarrow aB$$

# advanced grammar school

$A \rightarrow a$

$A \rightarrow aB$

$\text{Query} \rightarrow \text{Term}$

$\text{Query} \rightarrow \text{Term Query}$

$\text{Query} \rightarrow \text{Filter}$

$\text{Query} \rightarrow \text{Query Filter}$

advanced grammar school

$A \rightarrow a$

$A \rightarrow Ba$

$\text{Query} \rightarrow \text{Query Filter}$

$\text{Filter} \rightarrow \text{Modifier Entity}$



# advanced grammar school

$A \rightarrow a$

$A \rightarrow Ba$

$\text{Query} \rightarrow \text{Filter Query}$

$\text{Filter} \rightarrow \text{Modifier Entity}$

oh no



# advanced grammar school

A "context-free grammar"  
has rules that follow

$$A \rightarrow \alpha$$

where  $A$  is a non-terminal  
and  $\alpha$  is a combo of  
terminal and non-terminal

# advanced grammar school

$$S \rightarrow SS$$
$$S \rightarrow ()$$
$$S \rightarrow (S)$$
$$S \rightarrow []$$
$$S \rightarrow [S]$$

# advanced grammar school

```
<div class="Wrapper">  
  <input class="Input" />  
  <div class="Visualizer">  
    <div class="Token">Grammars!</div>  
  </div>  
</div>
```

# real world parsing

# real world: parsers

javascript	"front end"
------------	-------------

in:	#random
-----	---------

in:	#general
-----	----------

from:	@jenna
-------	--------

# real world: parsers

Term	Term
Modifier	Entity
Modifier	Entity
Modifier	Entity

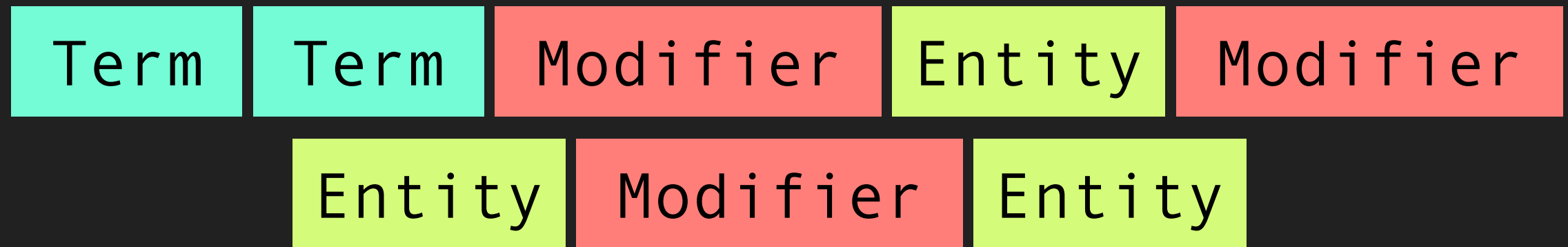
real world: parsers  
then, the parser goes  
through and matches the  
tokens to production  
rules

It's as if you know me better  
Than I ever knew myself  
I love how you can tell  
All the pieces, pieces, pieces of me  
♪ ♪ ♪

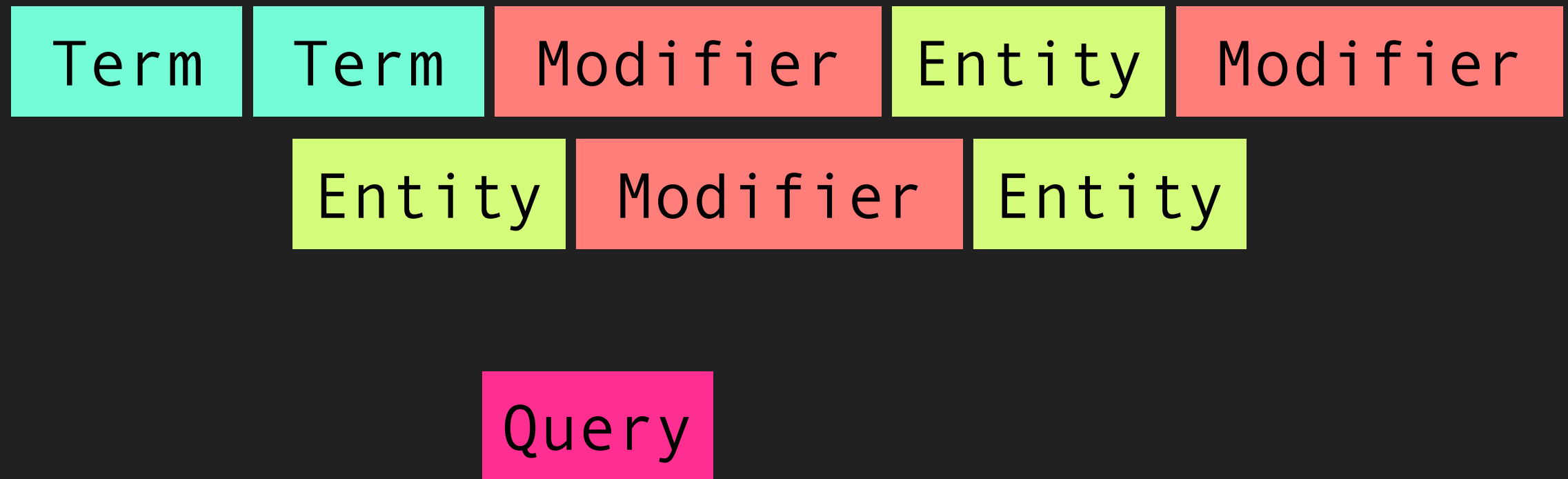




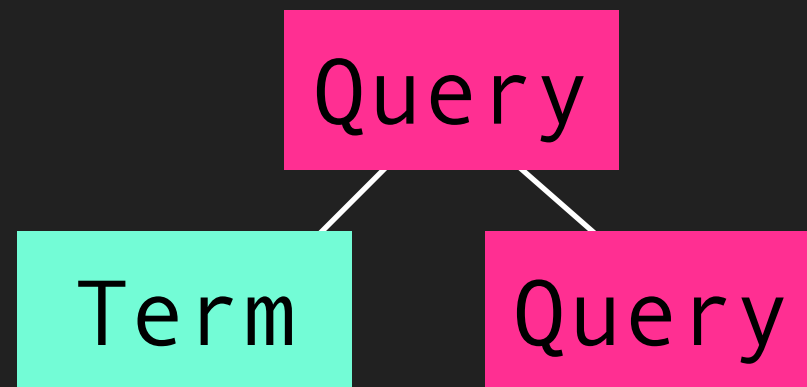
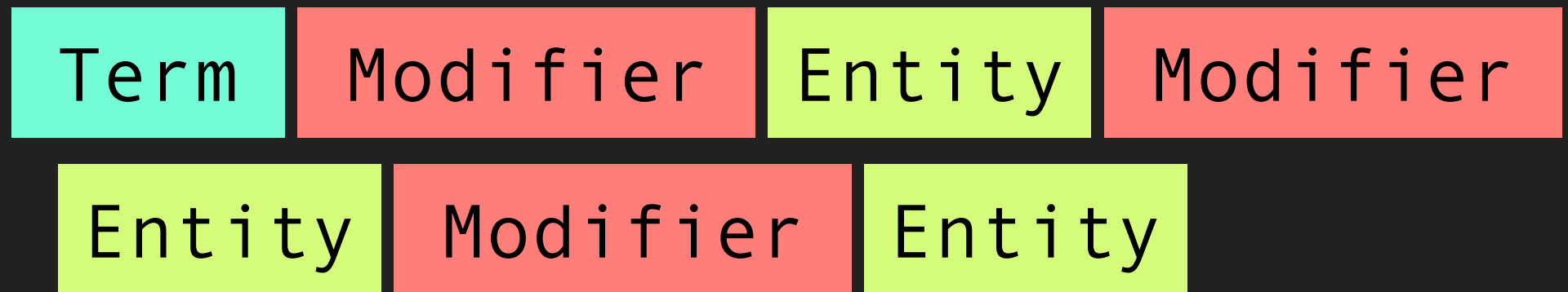
# real world: parsers



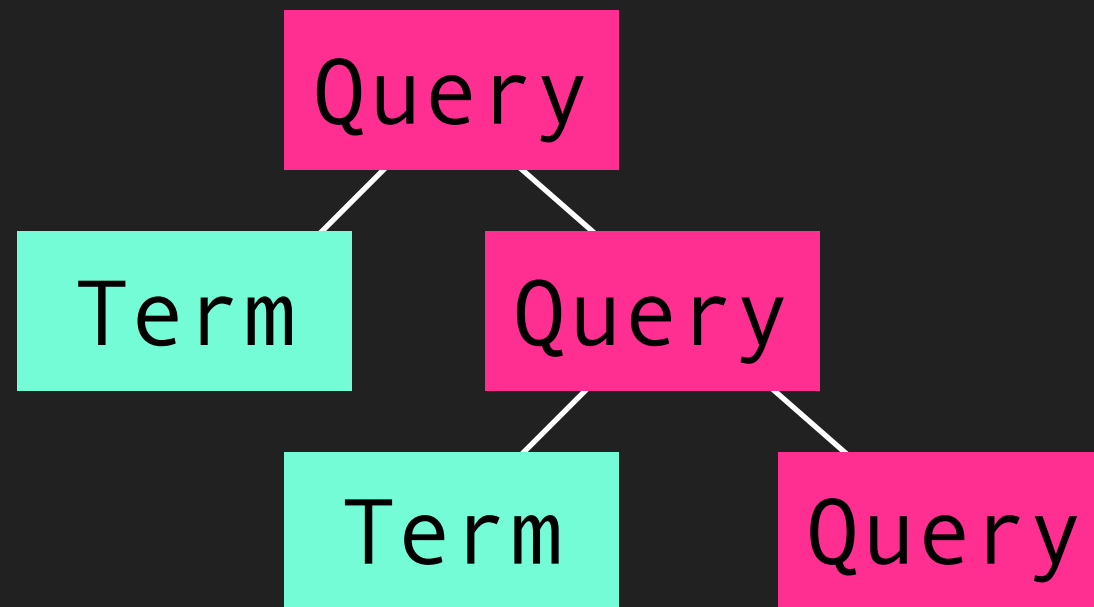
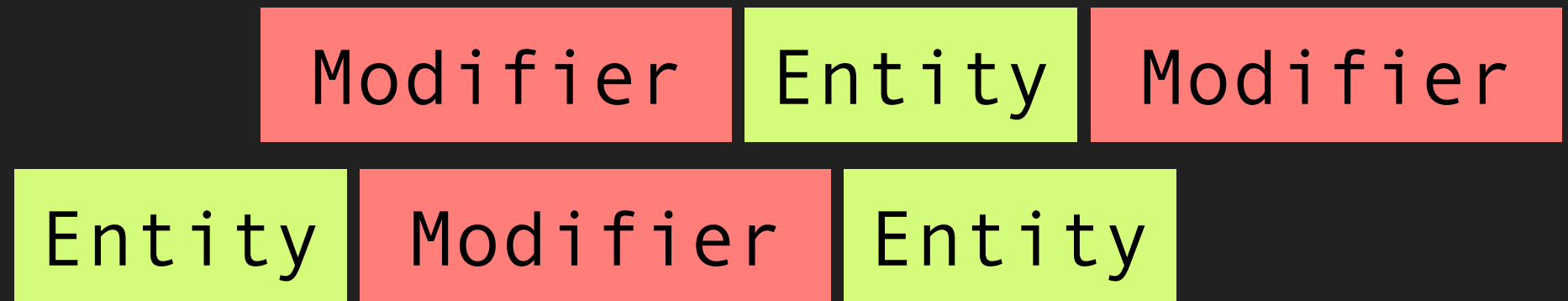
# real world: parsers



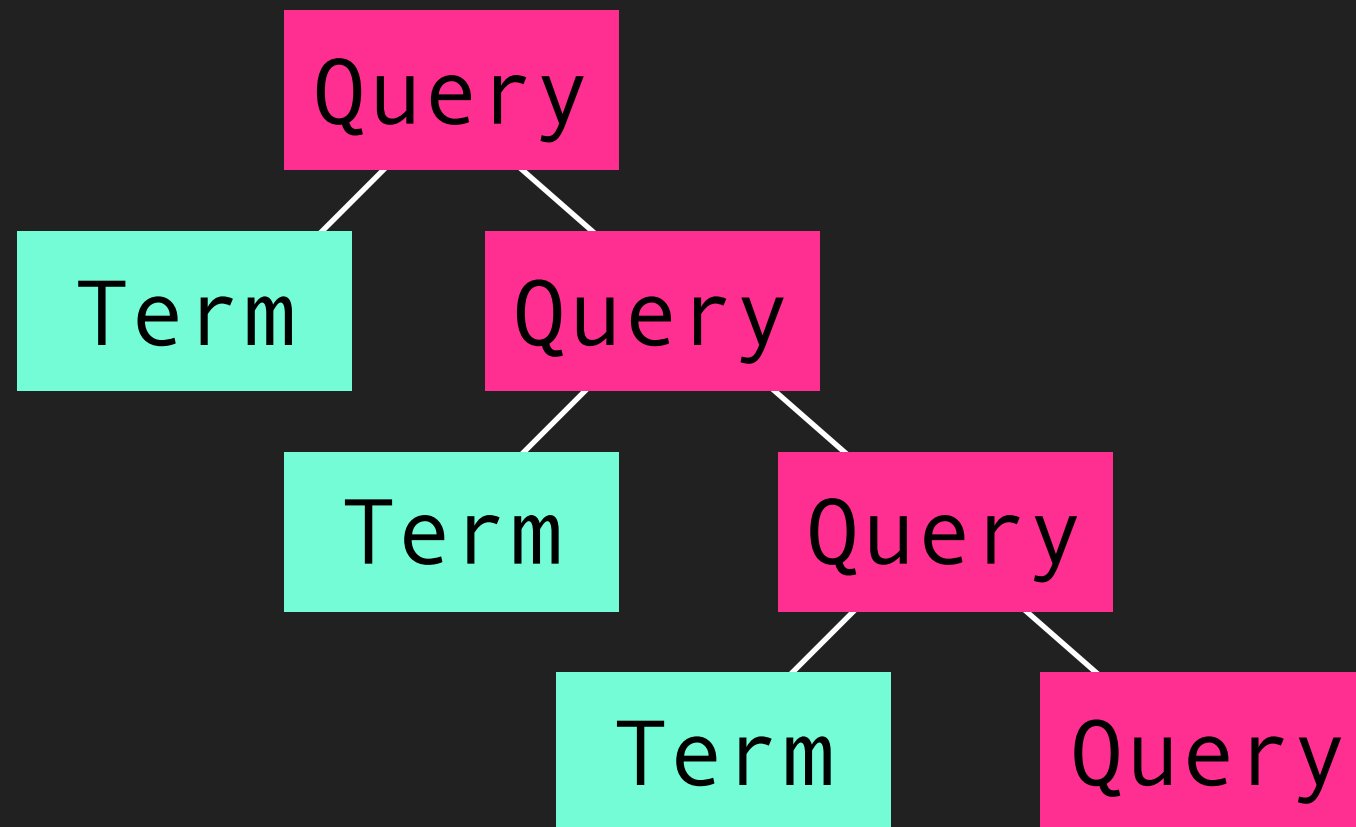
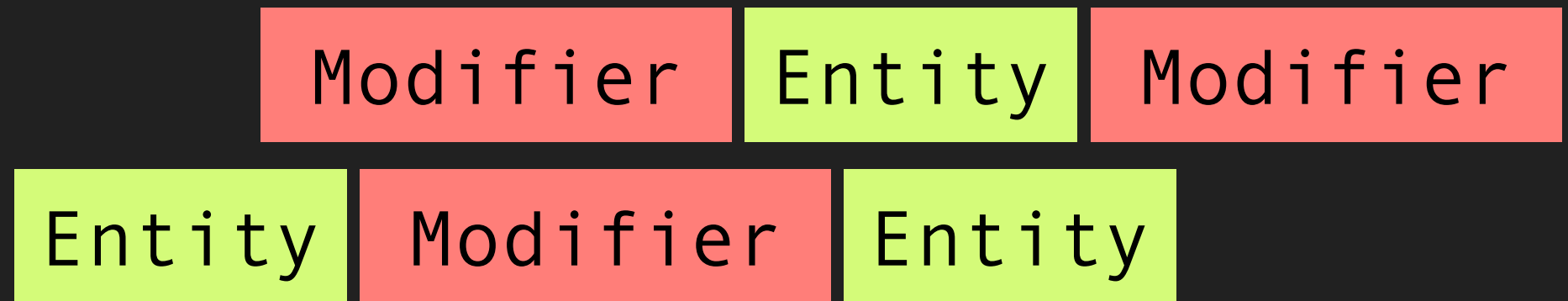
# real world: parsers



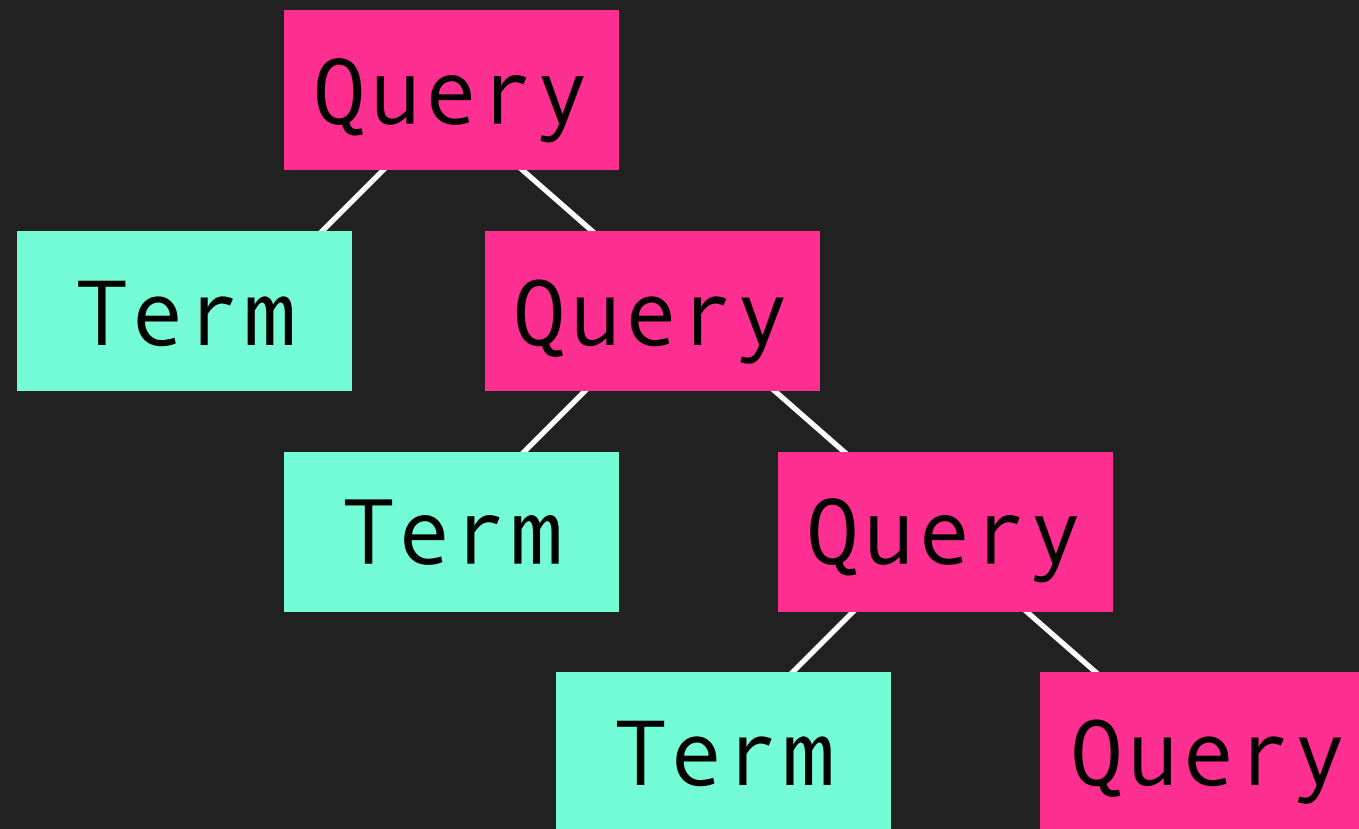
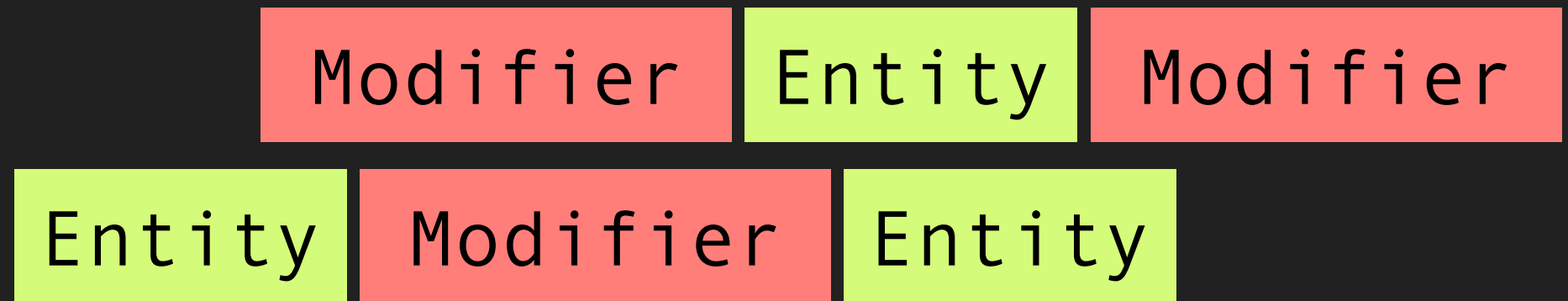
# real world: parsers



# real world: parsers



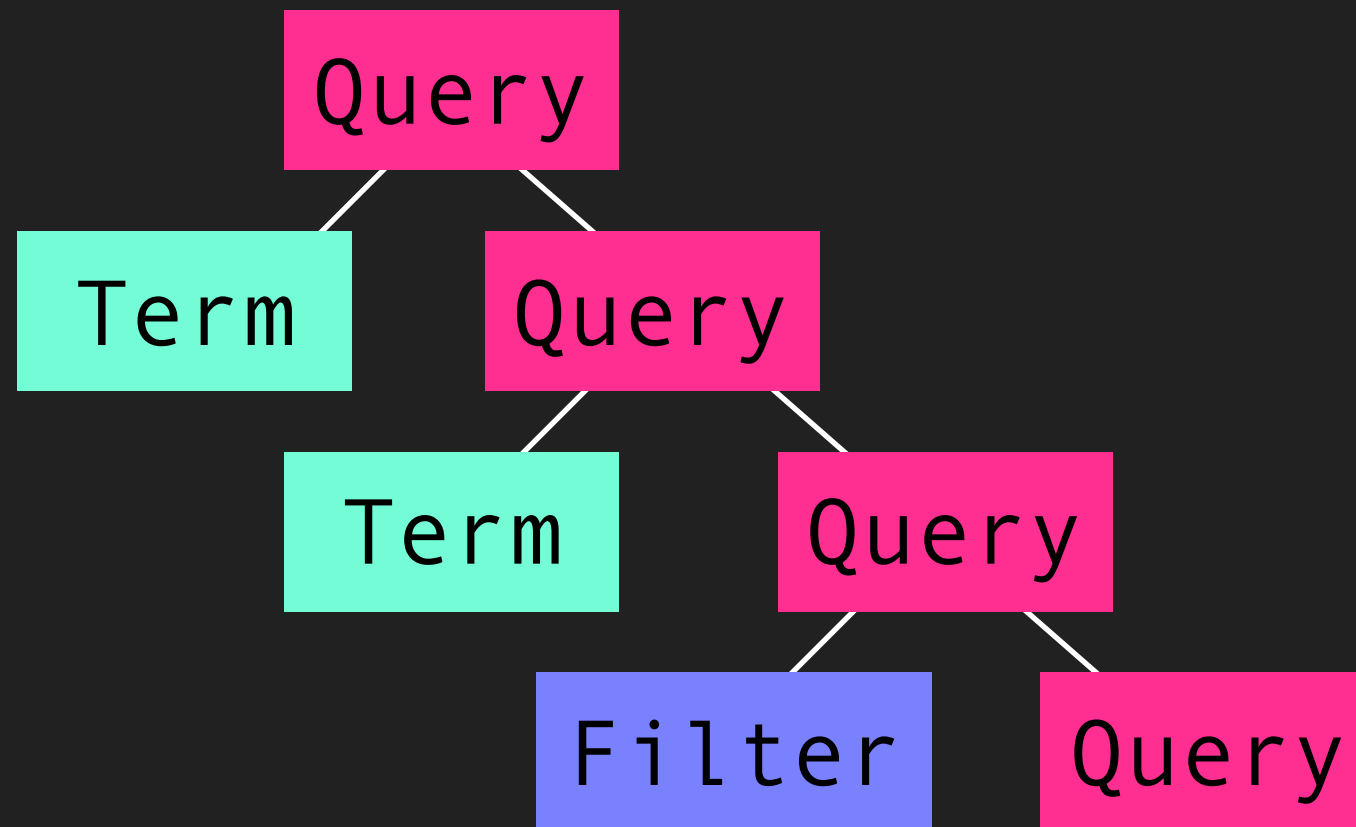
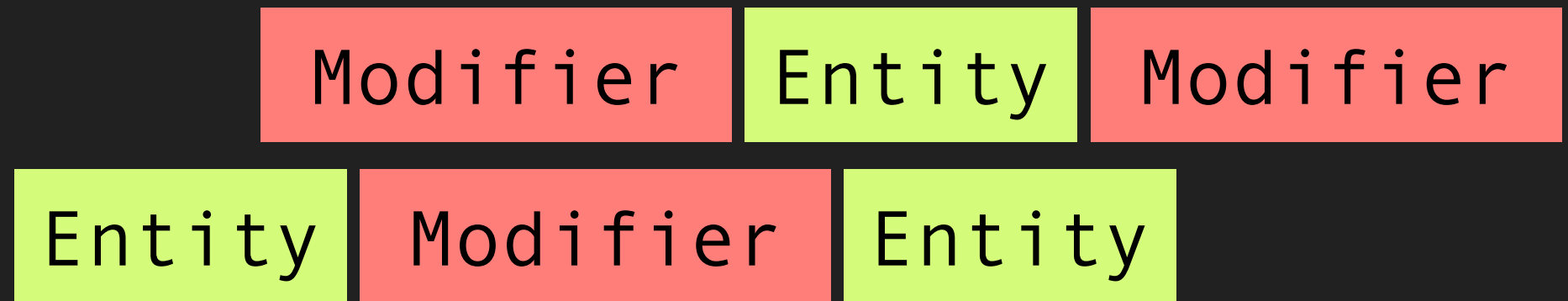
# real world: parsers



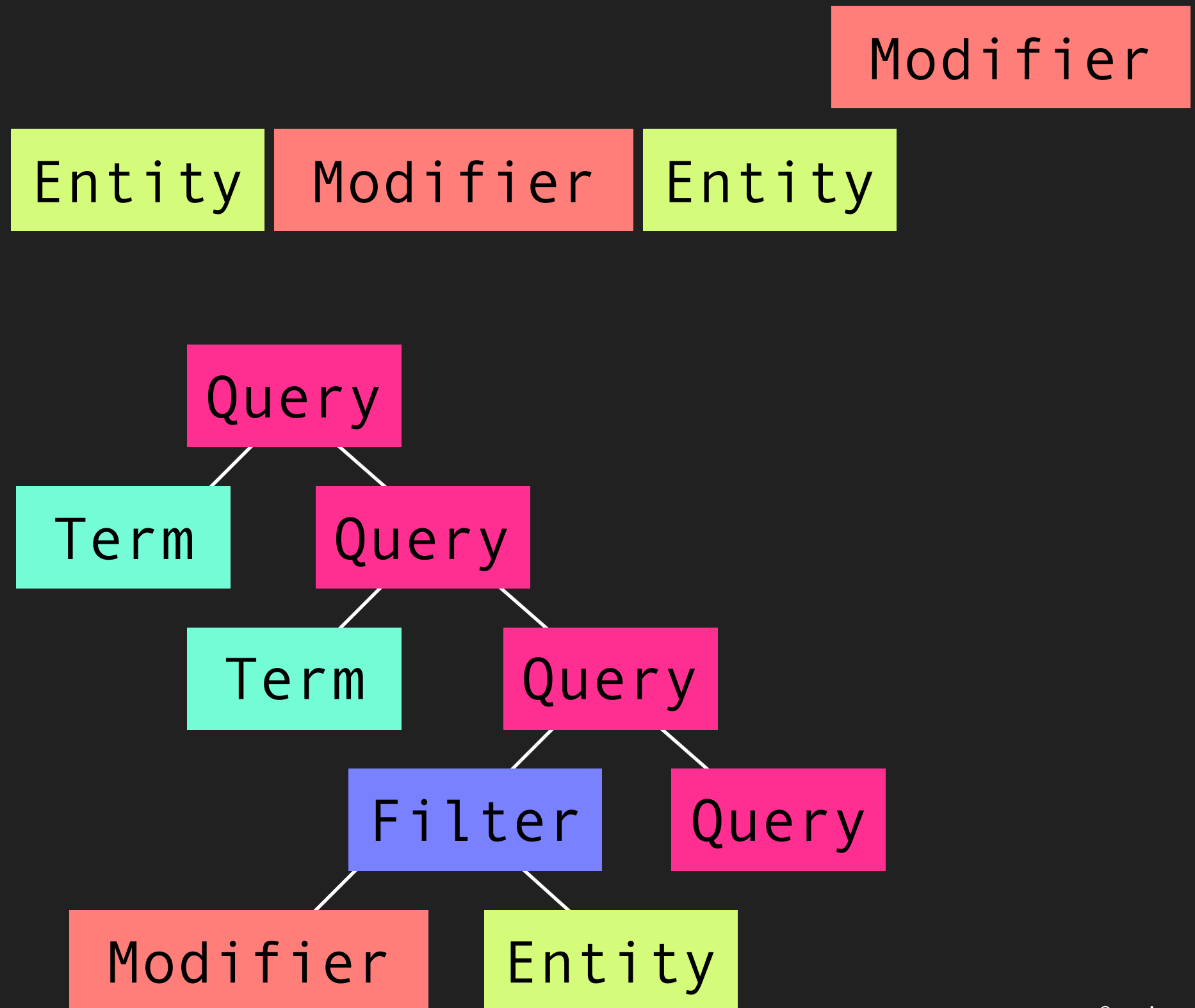
oh no



# real world: parsers



# real world: parsers





Grammars!

Lexers!

Tokens!

Parsers!

Trees!

"thank you"

"JSConf Hawaii"

from: @zeigenvector

jenna.is/at-jsconfhi

extra credit

the "Chomsky hierarchy"  
describes different  
classes of formal  
grammars

Stop! Grammar time!



MC Hammer - U Can't Touch This

[https://en.wikipedia.org/wiki/Noam\\_Chomsky](https://en.wikipedia.org/wiki/Noam_Chomsky)

[https://en.wikipedia.org/wiki/Chomsky\\_hierarchy](https://en.wikipedia.org/wiki/Chomsky_hierarchy)

@zeigenvector

jenna.is/at-jsconfhi

# extra credit

Type 0: Recursively Enumerable

Type 1: Context-Sensitive

Type 2: Context-Free

Type 3: Regular