Building parsers in JavaScript

Kenneth Geisshirt


kgeisshirt


kneth

Agenda

- What is parsing?
- Fractions
- Nearley
- Example: Fraction calculator

Rajiv Patel, https://tinyurl.com/yx6rmcdt

What is parsing?

You know the problem

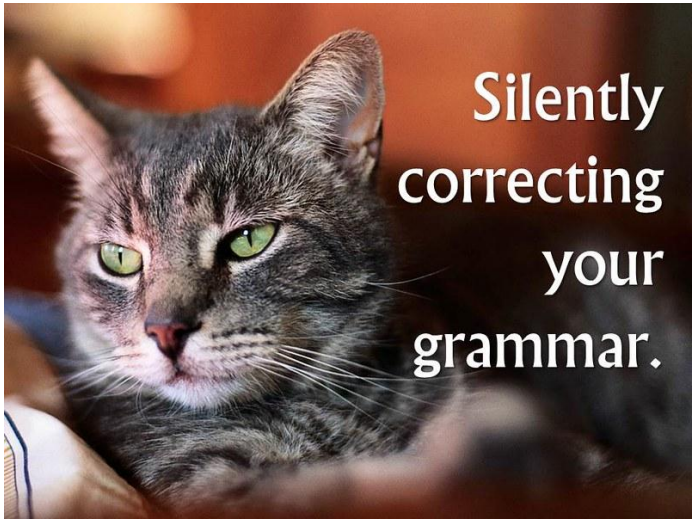
- What is syntax?
- How is the syntax defined?
- How do you check if input matches syntax?
- How can you use the syntax in your applications?

```
✔ ~
12:13 $ cat foobar.js
function add2(n) {
  return n + ;
}

const a = [1, 2, 3];
let b = a.map(n => add2(n));
console.log(b);

✔ ~
12:13 $ node foobar.js
/Users/kneth/foobar.js:2
  return n + ;
           ^


SyntaxError: Unexpected token ;
    at Module._compile (internal/modules/cjs/loader.js:723:23)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:789:10)
    at Module.load (internal/modules/cjs/loader.js:653:32)
    at tryModuleLoad (internal/modules/cjs/loader.js:593:12)
    at Function.Module._load (internal/modules/cjs/loader.js:585:3)
    at Function.Module.runMain (internal/modules/cjs/loader.js:831:12)
    at startup (internal/bootstrap/node.js:283:19)
    at bootstrapNodeJSCore (internal/bootstrap/node.js:623:3)
```

https://xkcd.com/859/

Grammar

- The syntax is defined by a grammar
- Lexical analysis breaks down input into tokens or terminals
  - Keywords, literals, identifiers, operators
- A set of rules connecting non-terminals to tokens
- One non-terminal is the start symbol
- Parsers are software which use a grammar to verify input



Ken Whytock, https://tinyurl.com/s9s3eee

```
function add2(n) {
  let r = n + 2;
  return r;
}
```

literal

identifier

keyword

Body: statements

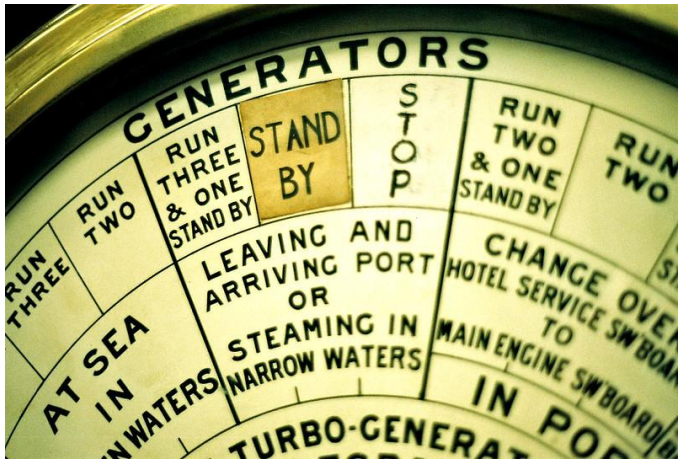
Example

- S → AA
- A → ***α***
- A → ***β***

Matches ***αα***, ***αβ***, ***βα***, and ***ββ***

Parser generators

- Many well-documented algorithms exist
  - Hot research topics in 1960s and 1970s
- It’s not a trivial task to write a parser
- Parser generators can speed up development process
  - Yacc (C) - 1975!!
  - ANTLR (mostly Java) - 1989
  - Nearley (JavaScript) - 2014



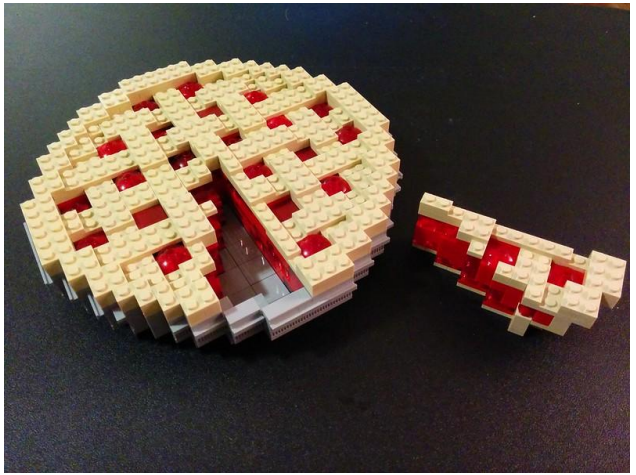
Erica Zabowski, https://tinyurl.com/uqbaldv

Fractions

Quick recap

- A fraction is a rational number
  - Numerator and denominator, both natural numbers
  - Broken latin (fractus, broken)
- Fractions are rational numbers

$$\frac{a}{b} \quad \text{where} \quad b \neq 0$$



Bill Ward, https://tinyurl.com/r3dtp2b

Arithmetic

$$f_1 + f_2 = f_2 + f_1$$

$$f_1 \cdot f_2 = f_2 \cdot f_1$$

$$\frac{a_1}{b_1} \pm \frac{a_2}{b_2} = \frac{a_1 \cdot b_2 \pm a_2 \cdot b_1}{b_1 \cdot b_2}$$

$$\frac{a_1}{b_1} \cdot \frac{a_2}{b_2} = \frac{a_1 \cdot a_2}{b_1 \cdot b_2}$$

$$\frac{a_1}{b_1} \Big/ \frac{a_2}{b_2} = \frac{a_1 \cdot b_2}{b_1 \cdot a_2}$$

$$\frac{n \cdot a}{n \cdot b} = \frac{a}{b}$$

Greatest Common Divisor

- Original algorithm by Euclid (c. 300 BC)
- Often used to reduce or simplify a fraction
- https://en.wikipedia.org/wiki/Greatest_common_divisor

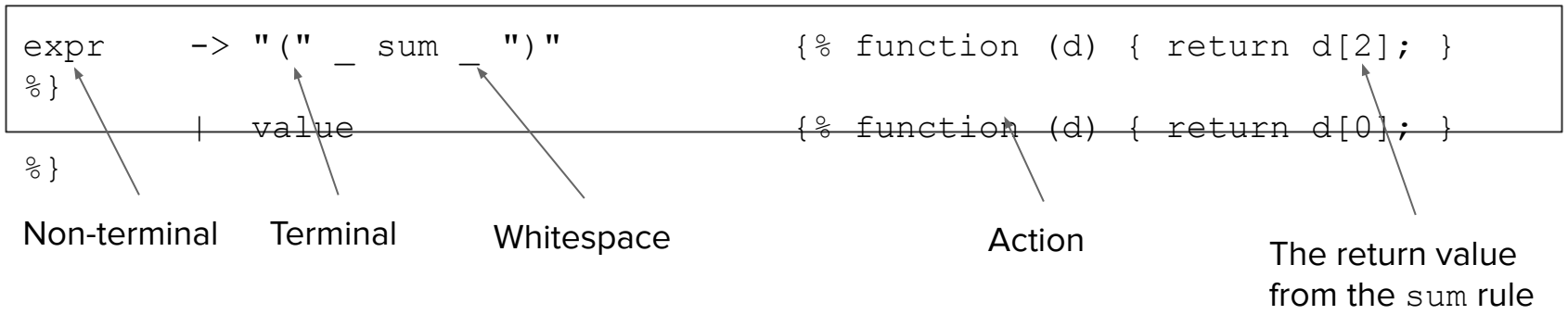
Nearley

Earley Parsers in JavaScript

- Nearley implements Earley's parser algorithm
  - Left-recursive (LR) grammars
  - Deterministic parser
  - Worst-case performance $O(n^3)$ but O(n) for well-behaving grammars
  - https://en.wikipedia.org/wiki/Earley_parser
- Can generate JavaScript, CoffeeScript, and TypeScript
  - Can run in browsers, node.js and probably React Native
- Inclusion of predefined grammars
  - Numbers, white spaces, strings
- Lexer is also included
  - Define tokens using double-quotes
- Rules can have (semantic) actions
  - Plain JavaScript functions

How to use

- Easy installation: `npm install nearley --save-dev`
- Generate a parser: `npx nearleyc -o parser.js parser.ne`
  - Add to `scripts` in `package.json`
- The `.ne` files contains rules, terminals, non-terminals, and actions

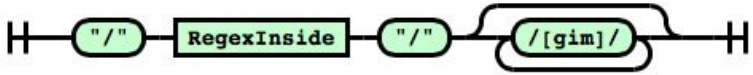


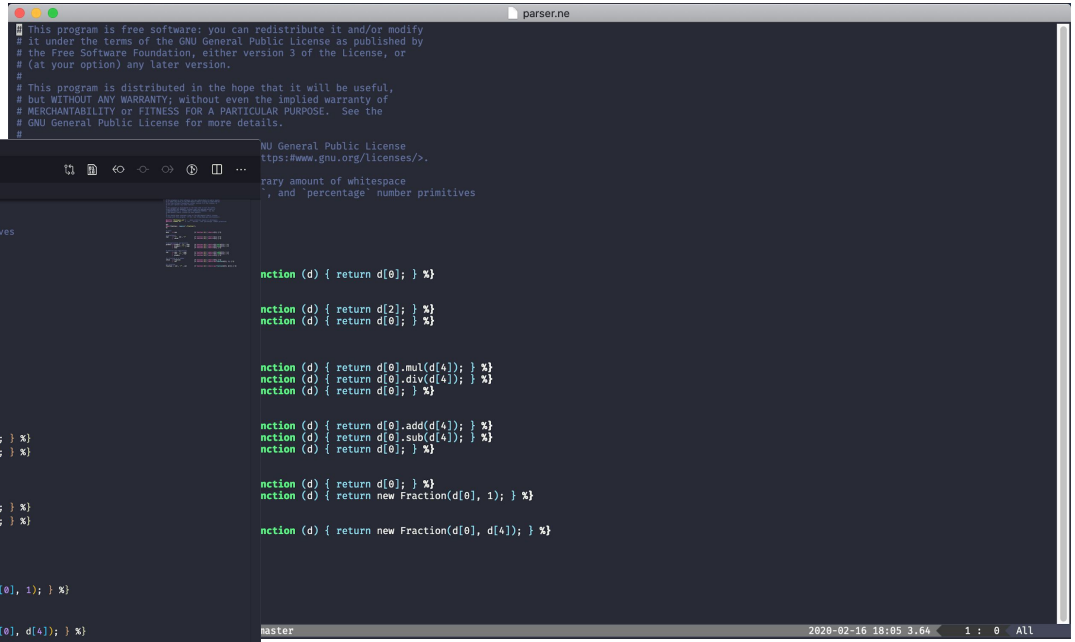
Additional tools

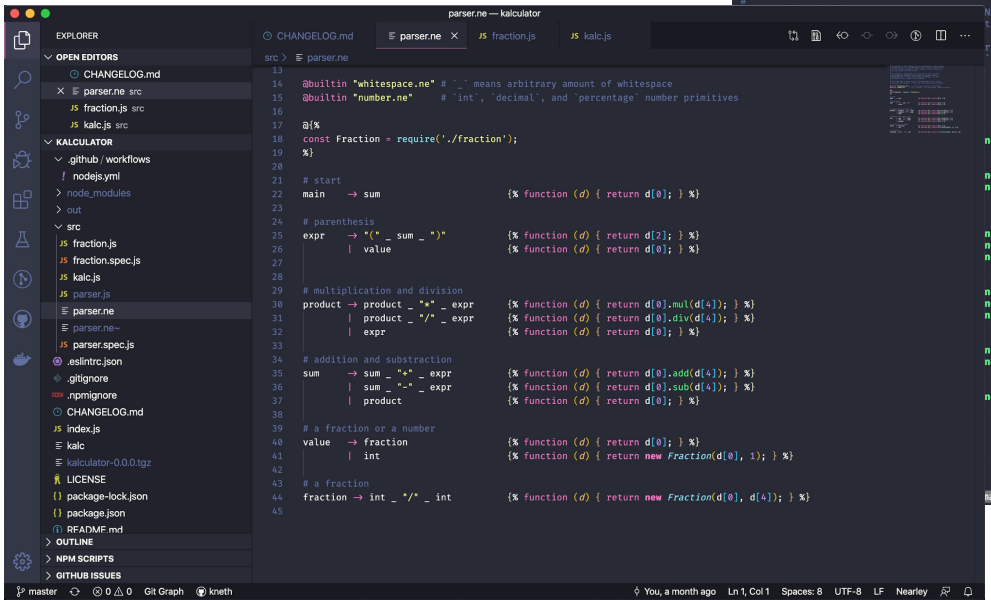
Supported by many editors

- VS Code, Atom, Emacs, Vim, Sublime

Railroad diagrams







Example: Fraction calculator

kalculator

- Simple Fraction class
  - Basic arithmetic and simplification
- Parser
  - Actions to perform calculation
- Little driver to read input and call parser

```
✔ ~/Projects/kalculator [master|✔]
17:29 $ ./kalc "2/3 * (5/7 + 8/17)"
2/3 * (5/7 + 8/17) = 94/119
✔ ~/Projects/kalculator [master|✔]
17:30 $ ./kalc "(2/3 + 9/2) * (7/8 + 1/2 + 3/4)"
(2/3 + 9/2) * (7/8 + 1/2 + 3/4) = 527/48
✔ ~/Projects/kalculator [master|✔]
17:30 $ ./kalc "(2/3 - 1/4) * (9/2 + 8/7 + 10/11) / (3/4 - 2/5)"
(2/3 - 1/4) * (9/2 + 8/7 + 10/11) / (3/4 - 2/5) = 25225/3234
```

The grammar (no actions)

```
main → sum
expr → ( sum )
     | value
product → product * expr
     | product / expr
     | value
sum → sum + expr
     | sum - expr
     | product
value → fraction
     | int
fraction → int / int
```

Start symbol: `main`
Tokens: `(`, `)`, `+`, `-`, `*`, `/`
Positive integer: `int`

Important take-aways

- Recursive rules
- Operator precedence

Source code: parser.ne

```
@builtin "whitespace.ne" # `_` means arbitrary amount of whitespace
@builtin "number.ne"     # `int`, `decimal`, and `percentage` number primitives

@{%
const Fraction = require('./fraction');
%}

# start
main    → sum                       {% function (d) { return d[0]; } %}

# parenthesis
expr    → "(" _ sum _ ")"           {% function (d) { return d[2]; } %}
        |  value                    {% function (d) { return d[0]; } %}


# multiplication and division
product → product _ "*" _ expr      {% function (d) { return d[0].mul(d[4]); } %}
        |  product _ "/" _ expr     {% function (d) { return d[0].div(d[4]); } %}
        |  expr                     {% function (d) { return d[0]; } %}

# addition and substraction
sum     → sum _ "+" _ expr          {% function (d) { return d[0].add(d[4]); } %}
        |  sum _ "-" _ expr         {% function (d) { return d[0].sub(d[4]); } %}
        |  product                  {% function (d) { return d[0]; } %}

# a fraction or a number
value   → fraction                  {% function (d) { return d[0]; } %}
        |  int                      {% function (d) { return new Fraction(d[0], 1); } %}

# a fraction
fraction → int _ "/" _ int          {% function (d) { return new Fraction(d[0], d[4]); } %}
```

Source code: kalc.js

```
const process = require('process');

const nearley = require("nearley");
const parser = require("./parser.js");

const p = new nearley.Parser(nearley.Grammar.fromCompiled(parser));
if (process.argv.length === 3) {
    const expr = process.argv[2];
    p.feed(expr);
    console.log(`${expr} = ${p.results[0].toString()}`);
} else {
    console.error(`Missing expression.`);
}
```

Resources

Links

- My example: https://github.com/kneth/kalculator
- Nearley: https://nearley.js.org/

- Earley parsers explained: http://loup-vaillant.fr/tutorials/earley-parsing/
- An Efficient Context-Free Parsing Algorithm. Jay Earley's Ph.D. thesis from 1968. http://reports-archive.adm.cs.cmu.edu/anon/anon/usr/ftp/scan/CMU-CS-68-earley.pdf

Building parsers for JavaScript is easy - and fun

Ron Mader, https://tinyurl.com/sg5pdwn