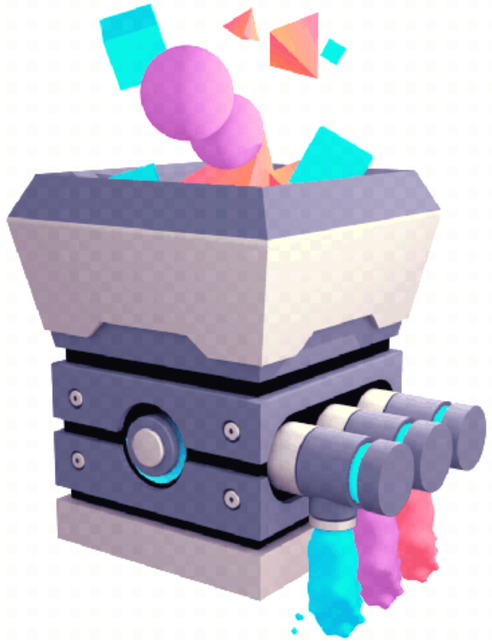

Halunke
.jetzt

INNOQ



Lucas

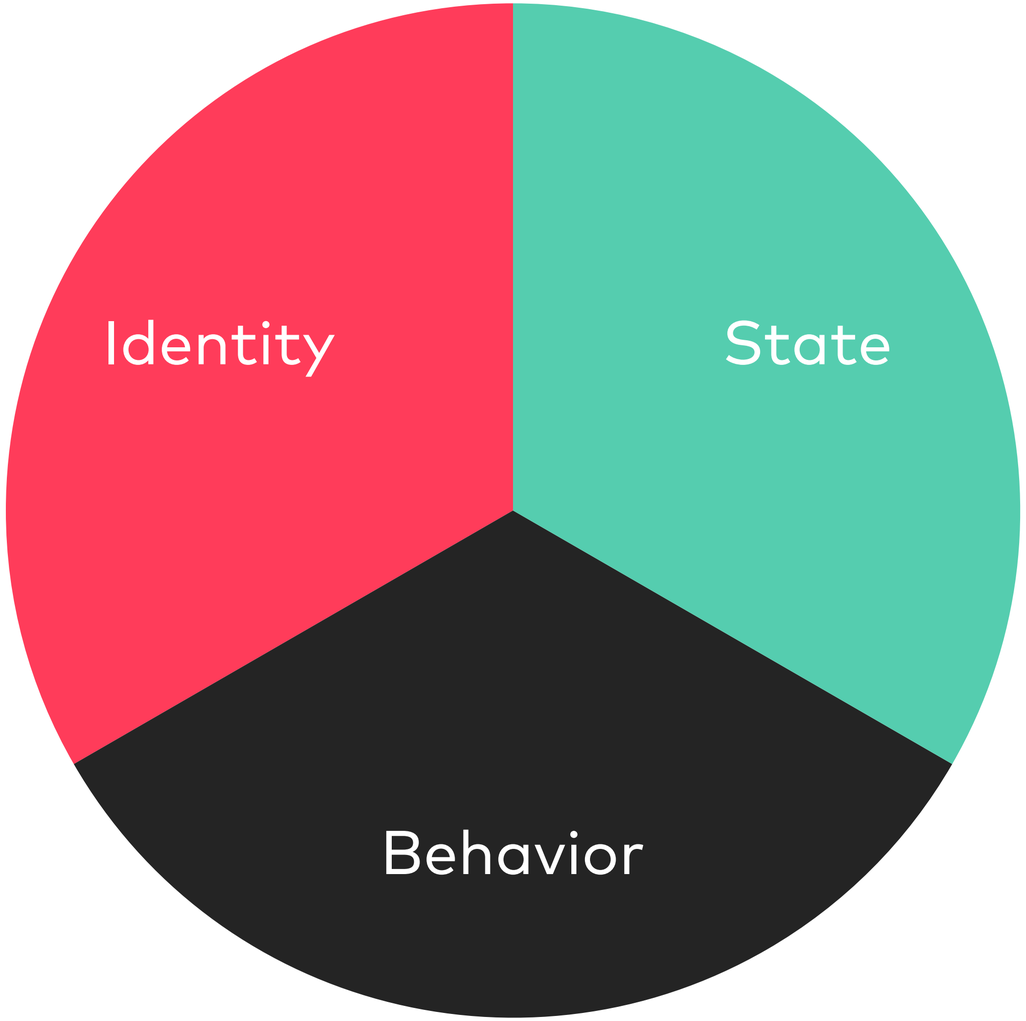
faucet-pipeline.org

complate

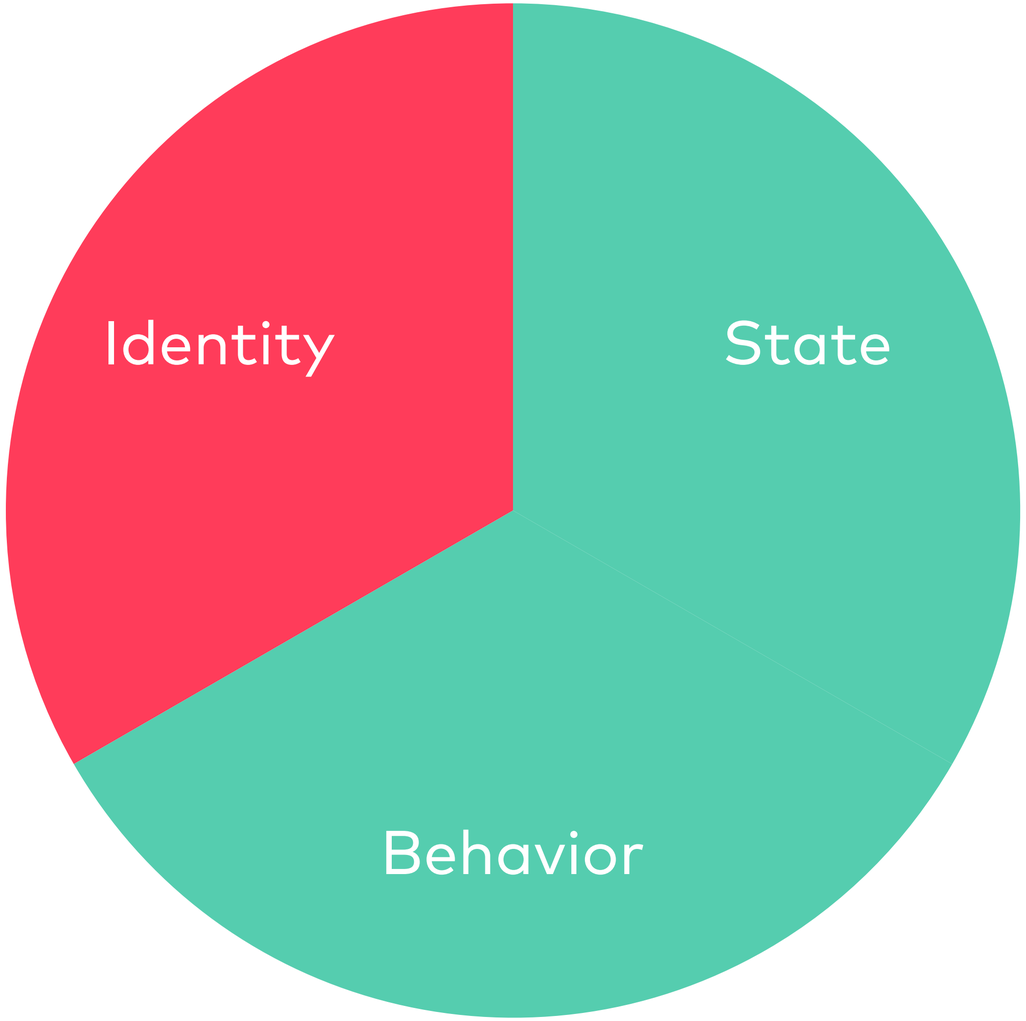
Part 1:
Motivation

"FP vs. OO"

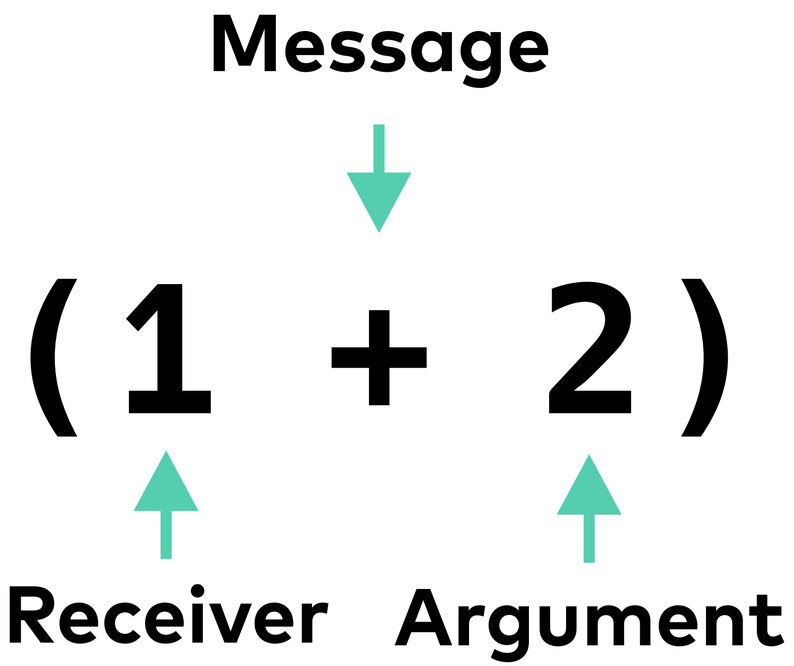
In Clojure...

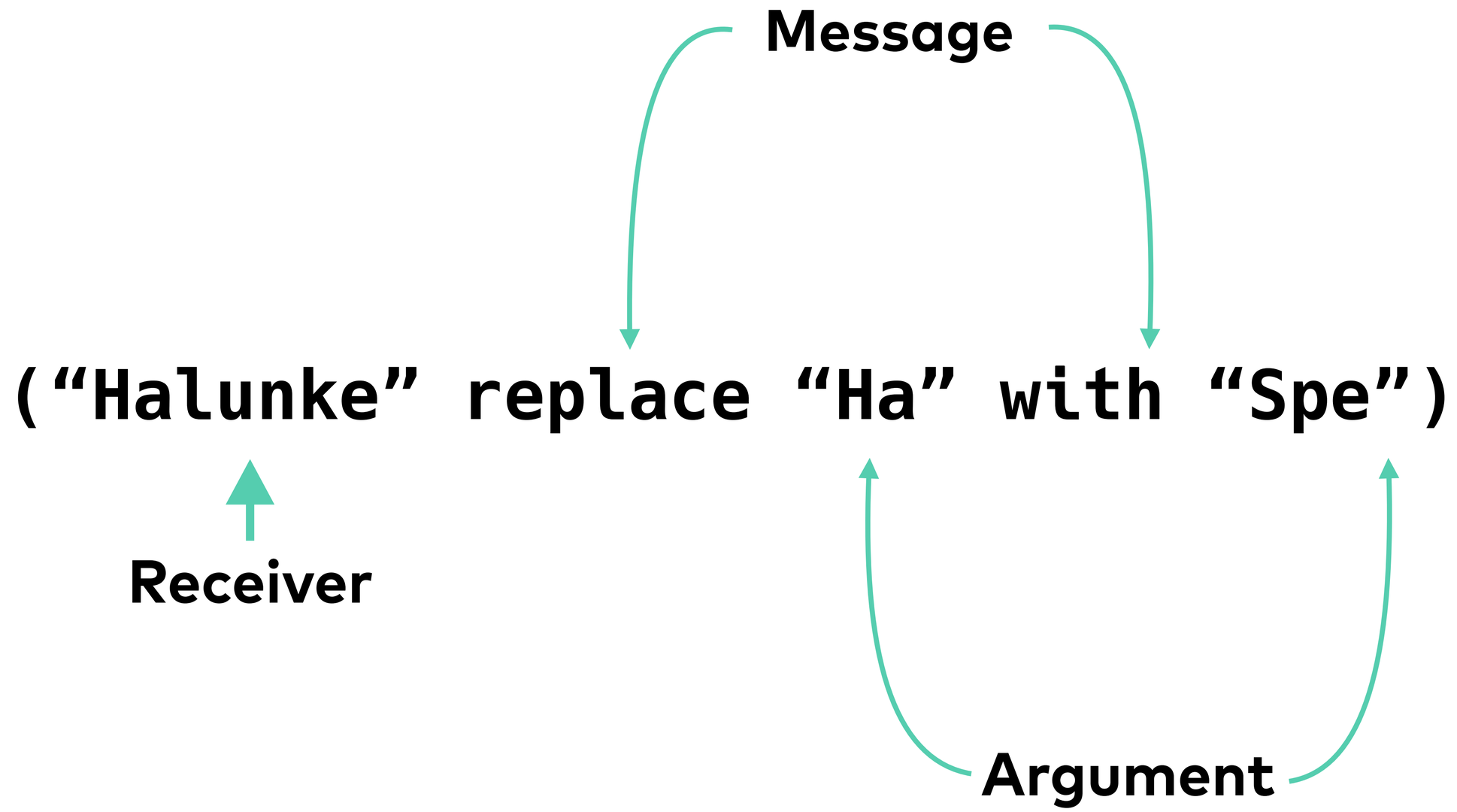


In Halunke...



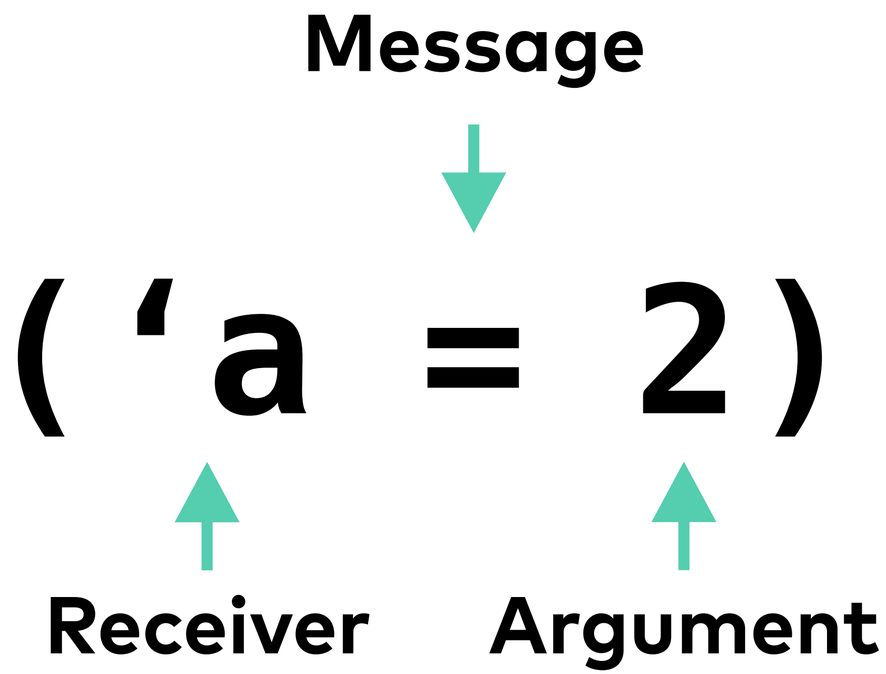
Part 2:
The Language


Message
(1 + 2)
Receiver
Argument


Message
("Halunke" replace "Ha" with "Spe")
Receiver
Argument

```
(“Halunke” replace “Ha” with “Spe”)
```

Sends “Halunke”
the message “replace with”
with the arguments [“Ha” “Spe”]


Message
(‘a = 2)
Receiver
Argument

- All objects are immutable
- A reference created with ‘a can’t be reassigned
- In the future, there will be reassignable references with @a

- ☑ No nil/null
- ☑ REPL
- ☑ Define own functions
- ☑ Define own classes
- ☑ Conditionals are messages, not syntax
- ☑ Number, String, Array, Dictionary, Regexp
- ☑ Web Module & a tiny Routing Lib

Currently in Development

```
('a = 12)
('my = "str")

("hello" replace (my replac "i" wit "a") with "mumpf")

(stdio puts "a")
```

```
4 | ("hello" replace (my replac "i" wit "a") with "mumpf")
                     ^^^^^^^^^^^^^^^^^^^^^^^^

The String "str" received the message `replac wit`. It doesn't know how to handle that.
Did you mean `replace with`?
```

Live Demo :)

Part 3: Implementation

Written in Ruby!

(“Halunke” replace “Ha” with “Spe”)

Lexer

```
a = Halunke::Lexer.new
a.tokenize(‘(“Halunke“ replace “Ha“ with “Spe“)‘)

⇒ [[:OPEN_PAREN, “(“],
    [:STRING, “Halunke“],
    [:BAREWORD, “replace“],
    [:STRING, “Ha“],
    [:BAREWORD, “with“],
    [:STRING, “Spe“],
    [:CLOSE_PAREN, “)“]]
```

Lexer (with Ragel)

1. Define Tokens

```
string = '"' [^"]* '"';
bareword = [a-zA-Z_]+ | '+' | '-' | '*' | '/' | '<' | '>' | '=' | '@';
open_paren = '(';
close_paren = ')';
```

Lexer (with Ragel)

2. React to tokens

```
string ⇒ { emit(:STRING, data[ts+1 ... te-1]) };
bareword ⇒ { emit(:BAREWORD, data[ts ... te]) };
open_paren ⇒ { emit(:OPEN_PAREN, data[ts ... te]) };
close_paren ⇒ { emit(:CLOSE_PAREN, data[ts ... te]) };
space;
any ⇒ { raise "Could not lex '#{ data[ts ... te] }'" };
```

Parser

```
b = Halunke::Parser.new
b.parse('("Halunke" replace "Ha" with “Spe")')
```

```
⇒ #<struct Halunke::Nodes nodes=[
    #<struct Halunke::MessageSendNode
      receiver=#<struct Halunke::StringNode value=“Halunke">,
      message=#<struct Halunke::MessageNode nodes=[
        #<struct Halunke::BarewordNode value=“replace">,
        #<struct Halunke::StringNode value=“Ha">,
        #<struct Halunke::BarewordNode value=“with">,
        #<struct Halunke::StringNode value=“Spe">
      ]>
    >
  ]>
```

Parser (with racc)

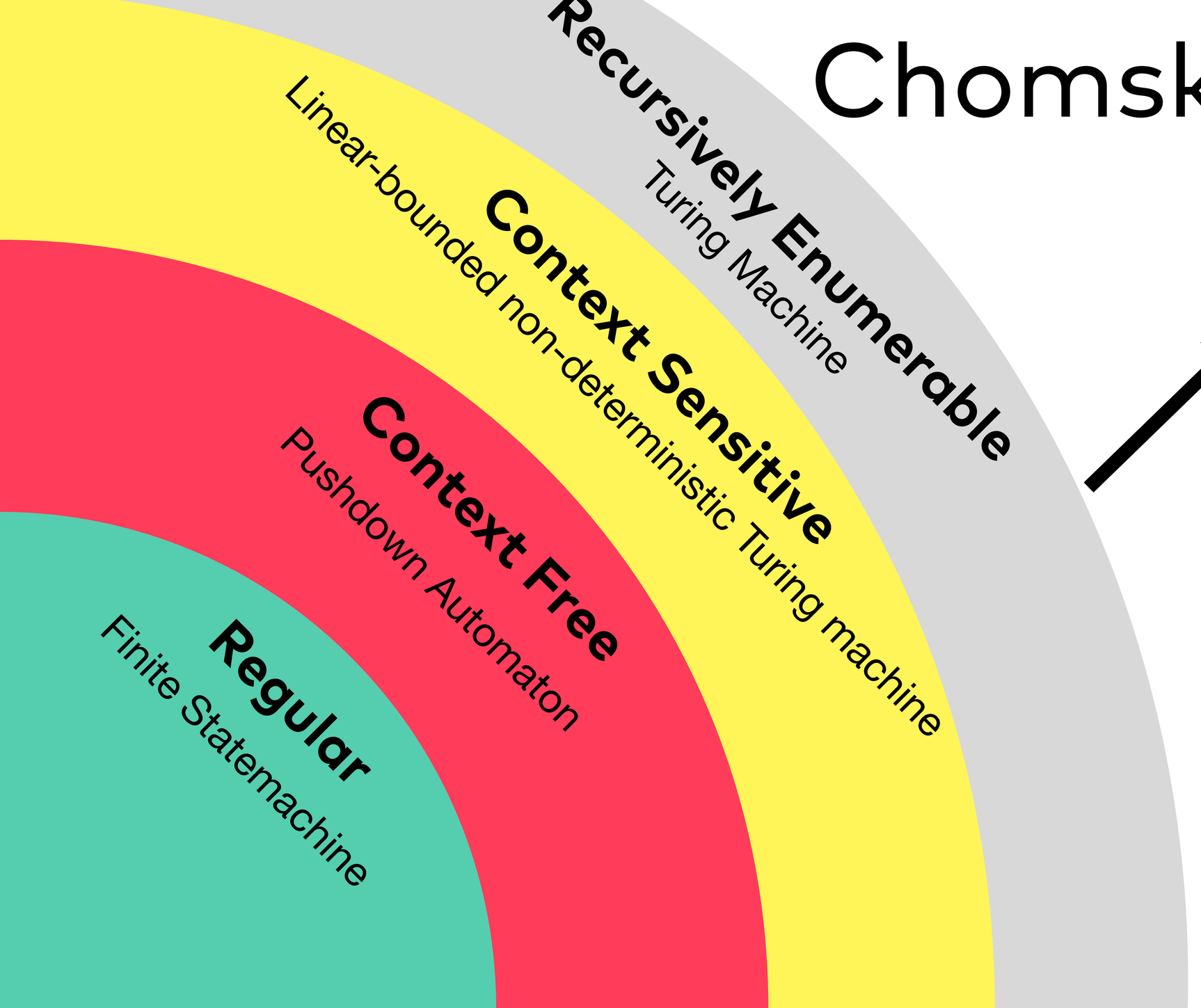
```
Program:
  Expressions { val[0] }
;

Expressions:
  /* empty */             { Nodes.new([]) }
| Expression Expressions { Nodes.new([val[0]].concat(val[1].nodes)) }
;

Expression:
  STRING                                          { StringNode.new(val[0]) }
| BAREWORD                                        { BarewordNode.new(val[0]) }
| OPEN_PAREN Expression Expressions CLOSE_PAREN   { MessageSendNode.new(val[1], val[2].to_message) }
;
```

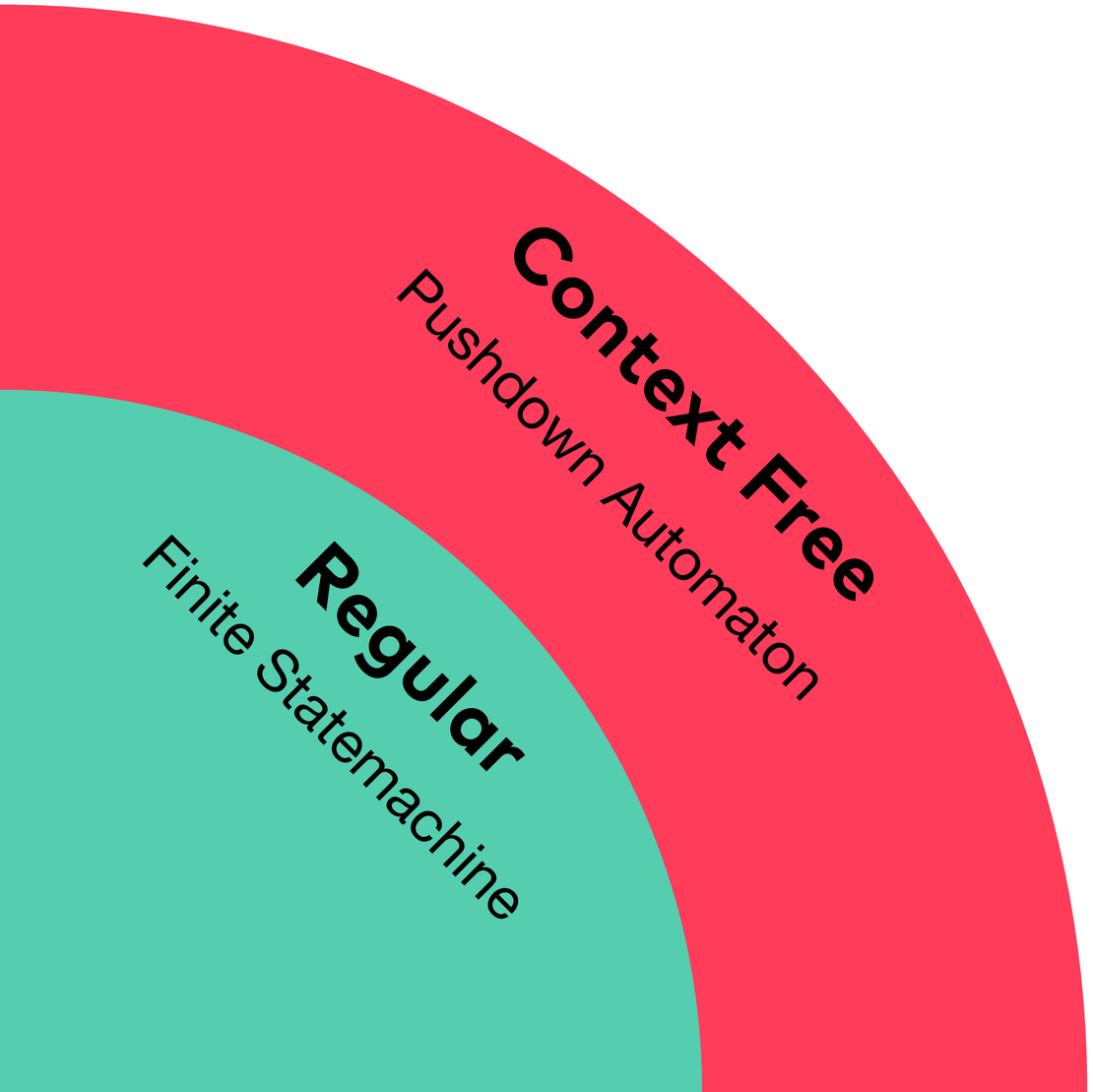
Why doesn't the parser just lex as well?

Chomsky hierarchy



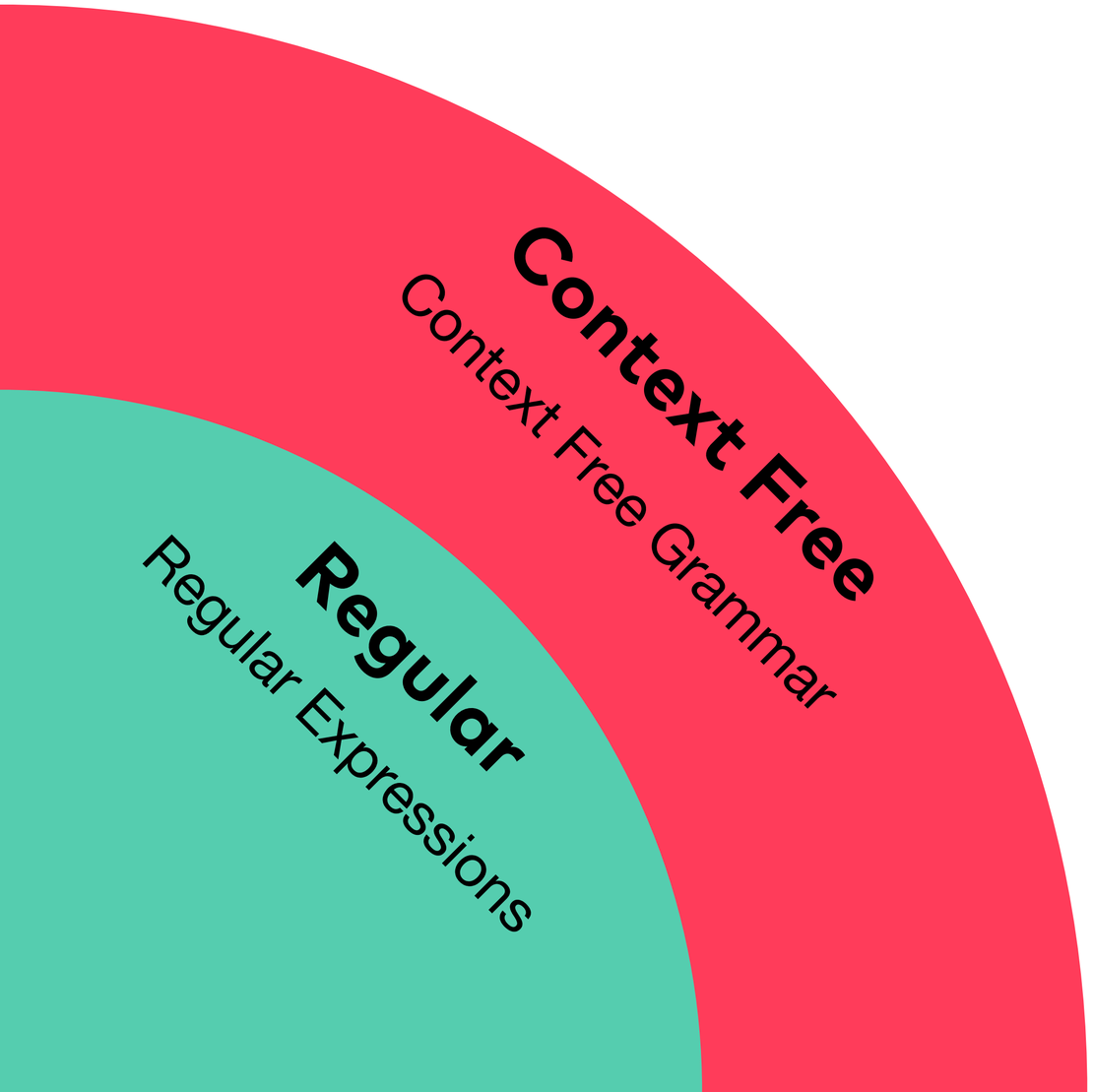
More powerful,
less guarantees,
less performance

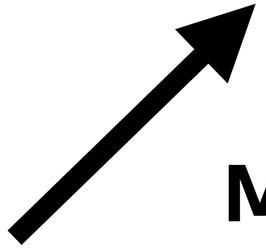
Chomsky hierarchy



More powerful,
less guarantees,
less performance

Chomsky hierarchy





More powerful,
less guarantees,
less performance

We have nodes now.
What's next?

Tree-Walk Interpreter

Every node has an eval method

```
Nodes = Struct.new(:nodes) do
  def eval(context)
    nodes.map { |node| node.eval(context) }.last
  end
end
```

What happens now?
Let's dive into the code

⌨️

https://try.halunke.jetzt

🐤

@moonbeamlabs