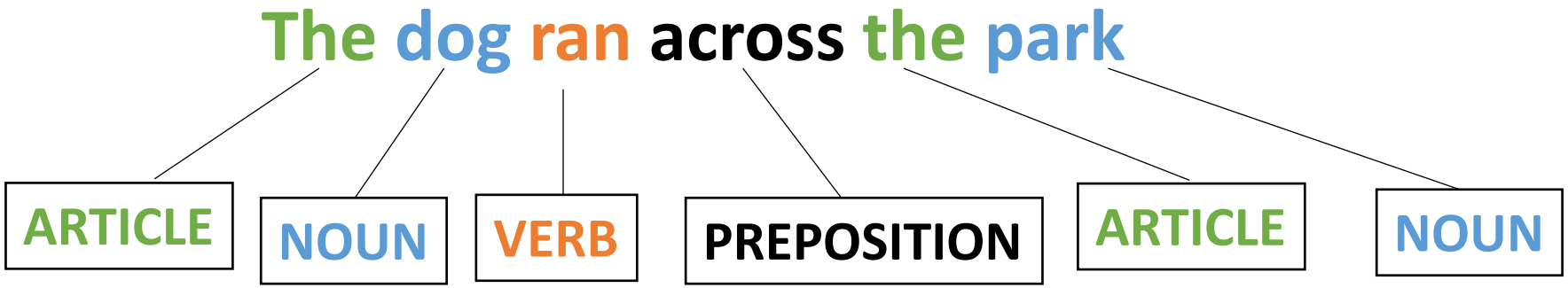
CSE110A: Compilers



- **Topics**:
  - *Lexical Analysis:*
    - Short comings of naïve scanner
  - ***Regular expressions**:*
    - Recursive definition
    - Syntactic sugar
    - groups

Review

Naïve implementation

- A scanner that implements

```
ID     = [characters]
NUM    = [numbers]
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = [“ “]
```

Naïve implementation

Building block:

```
class StringStream:
    def __init__(self, input_string):
        self.string = input_string


    def is_empty(self):
        return len(self.string) == 0


    def peek_char(self):
        if not self.is_empty():
            return self.string[0]
        return None


    def eat_char(self):
        self.string = self.string[1:]
```

Naïve implementation

Building block:

This class allows strings to be read as if we were doing I/O from a file.

So you are implementing with an abstraction that works both from a string or from a file.

```
class StringStream:
    def __init__(self, input_string):
        self.string = input_string

    def is_empty(self):
        return len(self.string) == 0

    def peek_char(self):
        if not self.is_empty():
            return self.string[0]
        return None

    def eat_char(self):
        self.string = self.string[1:]
```

Naïve implementation

First step in implementing the scanner

```
class NaiveScanner:

    def __init__(self, input_string):
        self.ss = StringStream(input_string)

    def token(self):

        while self.ss.peek_char() in IGNORE:
            self.ss.eat_char()

        if self.ss.is_empty():
            return None
```

```
ID     = [characters]
NUM    = [numbers]
ASSIGN = "="
PLUS   = "+"
MULT   = "*"
IGNORE = [" "]
```

Naïve implementation

First step in implementing the scanner

```
class NaiveScanner:

    def token(self):
        ...
        if self.ss.peek_char() == "+":
            value = self.ss.peek_char()
            self.ss.eat_char()
            return ("ADD", value)

        if self.ss.peek_char() == "*":
            value = self.ss.peek_char()
            self.ss.eat_char()
            return ("MULT", value)
```

```
ID     = [characters]
NUM    = [numbers]
ASSIGN = "="
PLUS   = "+"
MULT   = "*"
IGNORE = [" "]
```

Naïve implementation

First step in implementing the scanner

```
class NaiveScanner:

    def token(self):
        ...
        if self.ss.peek_char() in NUMS:
            value = ""
            while self.ss.peek_char() in NUMS:
                value += self.ss.peek_char()
                self.ss.eat_char()
            return ("NUM", value)
```

```
ID     = [characters]
NUM    = [numbers]
ASSIGN = "="
PLUS   = "+"
MULT   = "*"
IGNORE = [" "]
```

Topics:

- Naïve Parser:
  - Code demo and discussion
- Regular expressions

Code Demo

Shortcomings of Naïve scanner

- Any thoughts?

Shortcomings of Naïve scanner

- IDs with numbers in them?
  - `x1, y1, etc.`
  - how would you solve?
- Numbers with a decimal point in them?
  - `4.5, 9999.99998`
  - how would you solve this?
- Two character operators:
  - `++, +=`
  - how would you solve this?

Shortcomings of Naïve scanner

- IDs with numbers in them?
  - `x1, y1, etc.`
  - how would you solve?
- Numbers with a decimal point in them?
  - `4.5, 9999.99998`
  - how would you solve this?
- Two character operators:
  - `++, +=`
  - how would you solve this?

Things get really hacky really quickly!

Creates a bad design that is not easily extended or maintained

How do we solve this?

A new token definition language:

- **Regular Expressions (RE)**
- Tokens will be defined using regular expressions
- Scanners can then utilize regular expression matchers

Benefits:

- Extensible design
  - easy to add new tokens, modify existing definitions
- Modular
  - Scanner can utilize common regex libraries

Cons:

How do we solve this?

A new token definition language:

- **Regular expressions**
- Tokens will be defined using regular expressions
- Scanners can then utilize regular expression matchers

Benefits:

- Extensible design
  - easy to add new tokens, modify existing definitions
- Modular
  - Scanner can utilize common regex libraries

Cons:

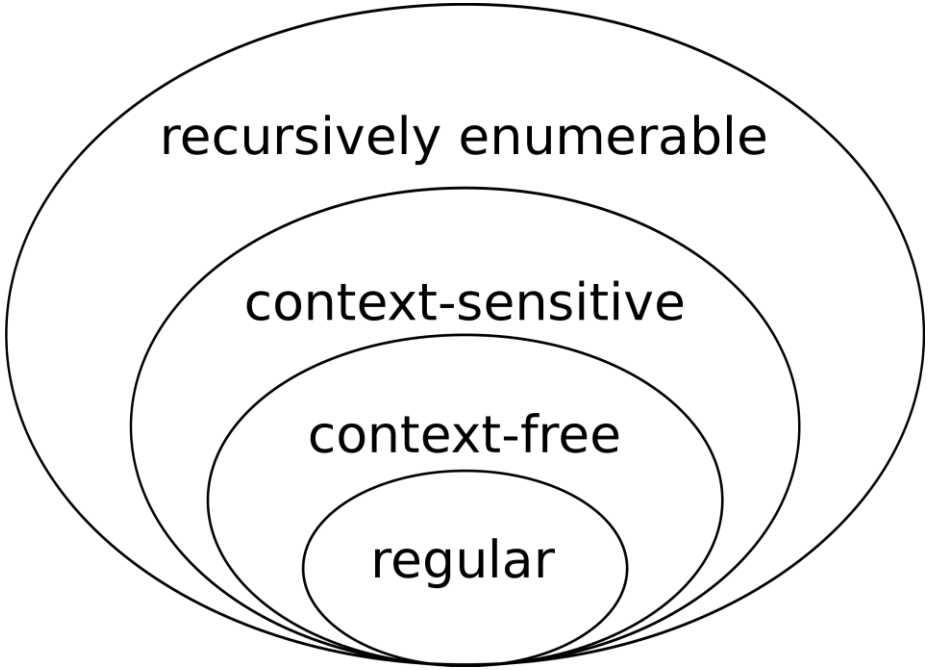
- Token definitions are restricted to regular languages
- Potentially slower
- Regular expression matchers are complicated

Regular expressions

Some theory:

- Given a language *L*, a string *s* is either part of that language or not
  - Integers are a language: “5”, “6”, “-7” is in the language. “abc” is not.
- Languages are grouped into families depending on how “hard” it is to determine if a string is part of that language.

Regular expressions



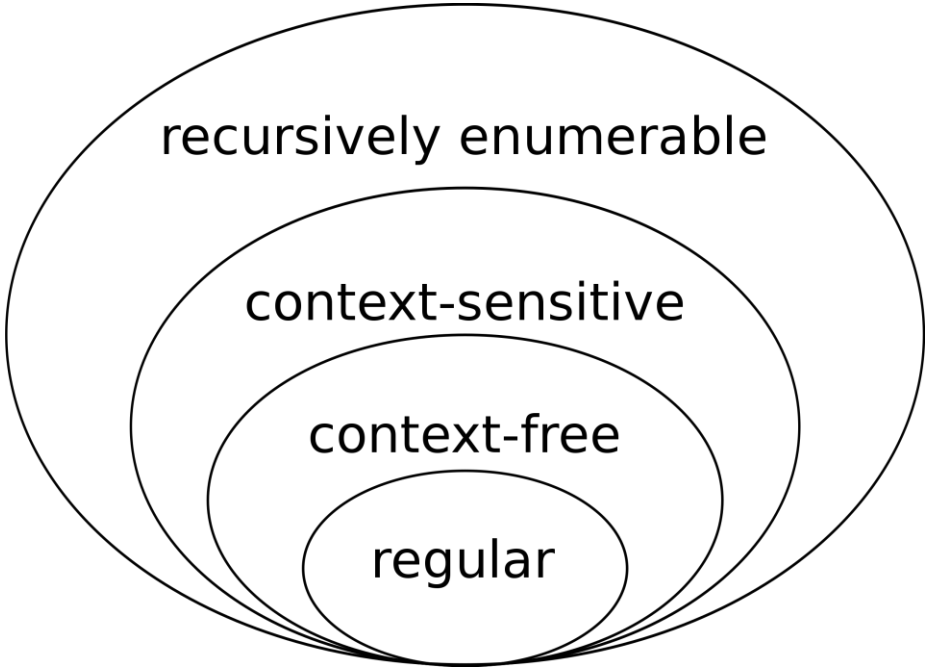
The simplest languages are regular. We will use regular languages as our token language.

We will use the next level: context-free, as the language for our parser.

Higher levels are interesting, but not as useful in compilers. Why?

image source: wikipedia

Regular expressions



Language Types according to
the Noam Chomsky Hierarchy

The simplest languages are regular. We will use regular languages as our token language.

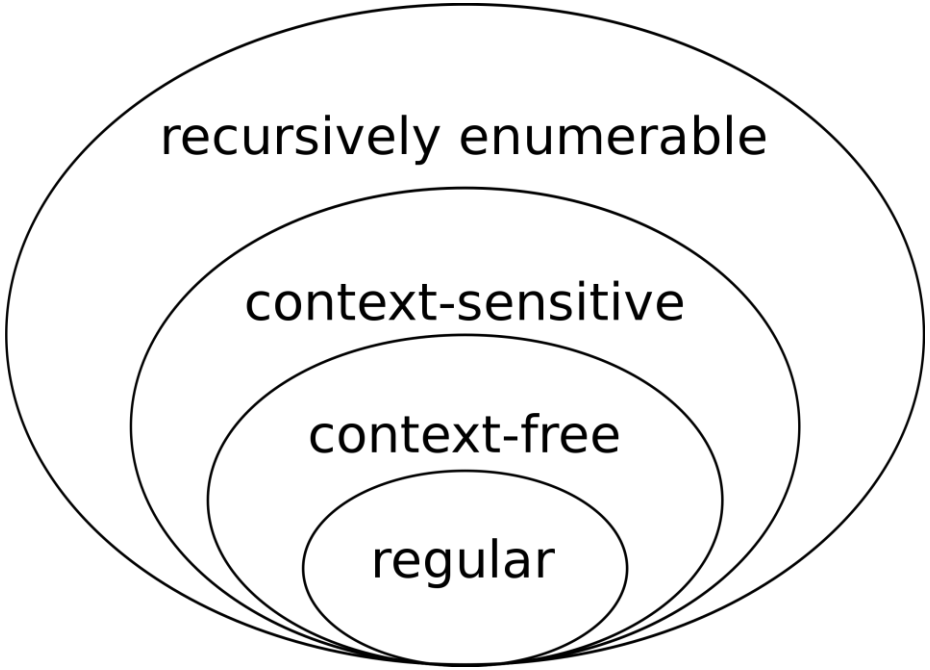
We will use the next level: context-free, as the language for our parser.

Higher levels are interesting, but not as useful in compilers. Why?

Because deciding if a string is in a recursively enumerable language is undecidable.

image source: Wikipedia: https://en.wikipedia.org/wiki/Chomsky_hierarchy

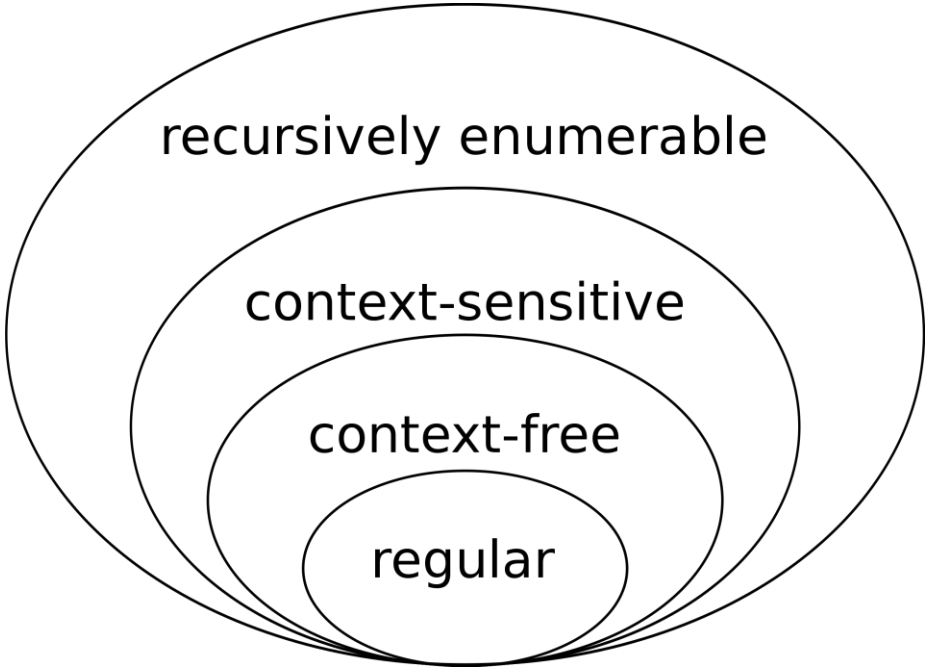
Regular expressions



What is a regular language?

image source: wikipedia

Regular expressions

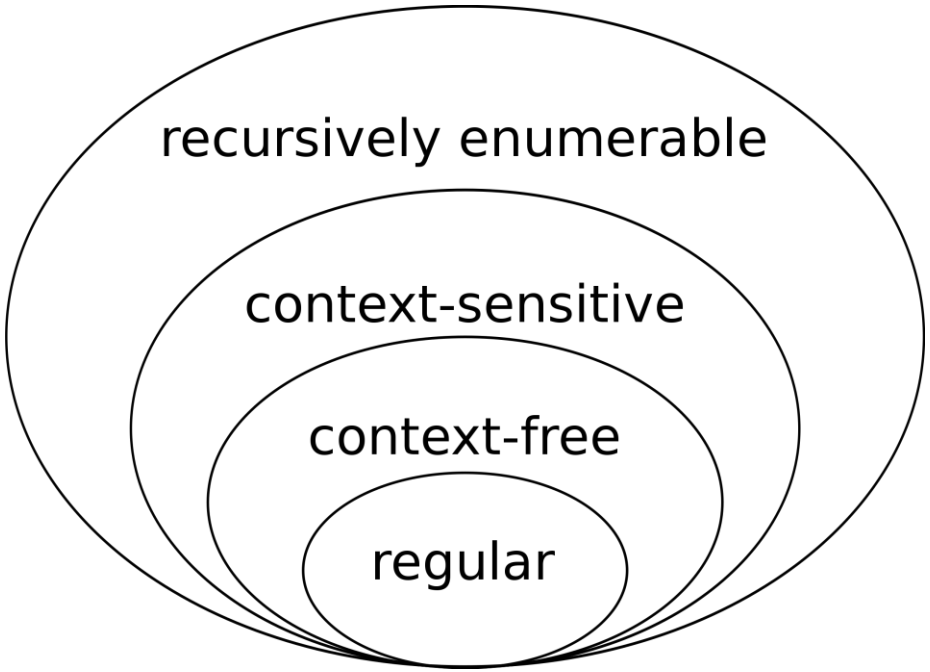


What is a regular language?

For this class: *A regular language is a language that can be expressed as a regular expression.*

image source: wikipedia

Regular expressions



What is a regular language?

For this class: *A regular language is a language that can be expressed as a regular expression.*

What is a regular expression?

image source: wikipedia

Regular expressions

- We will define regular expressions (RE) recursively
- We will show examples at each step.
- And show to match them in Python
  - *A string matches an RE if it belongs to the regular language defined by the RE*
  - Python has a great RE matching library

Regular expressions in Python

```
# import the library
import re

# pattern is a string representing the RE
# the function reports whether string matches RE
re.fullmatch(pattern, string)
```

Regular expressions

- **We will define regular expressions (RE) recursively**

- Like any recursive function, we can start with the base case:

a regular expression can be a single character or the empty string

Regular expressions

- **We will define regular expressions (RE) recursively**

- Like any recursive function, we can start with the base case:

a regular expression can be a single character or the empty string

Example:

```
ASSIGN = “=“
PLUS   = ”+”
```

Python:

```
import re
re.fullmatch(“=“, “=“)

re.fullmatch(“+”, “+”) # what happens here?
```

Regular expressions

- When we define regular expressions, some characters are special.
  - They are operators in the regular expression language
  - If we want to use them as a character, then we need to ”escape them” with a \
  - “+” happens to be one of those characters

https://riptutorial.com/regex/example/15848/what-characters-need-to-be-escaped-

Python:

```
import re
re.fullmatch(“=“, “=“)

re.fullmatch(“\+”, “+”) # what happens here?
```

Regular expressions

- **We will define regular expressions (RE) recursively**

- Like any recursive function, we can start with the base case:

  a regular expression can be a single character or the empty string

Python:

```
import re
re.fullmatch("", "")
```

Not super useful for us, but useful for the theory

Regular expressions

- First recursive case: **concatenation**

- Two REs can be concatenated by simply writing them in sequence:
  - RE1 = ”a”, RE2 = “b”
  - concatenated it is: RE12 = “ab”

- This allows us to build words

Example:

```
FOR   = “for“
WHILE = ”while”
```

Python:

```
import re
re.fullmatch(“for“, “for“)
re.fullmatch(“a+b”, “a+b”) # what happens here?
```

Can we define these tokens yet?

Tokens		Tokens Definitions
• ARTICLE	=	{The, A, My, Your}
• NOUN	=	{Dog, Car, Computer}
• VERB	=	{Ran, Crashed, Accelerated}
• ADJECTIVE	=	{Purple, Spotted, Old}

Can we define these tokens yet? No, we need one more operator

Tokens		Tokens Definitions
• ARTICLE	=	{The, A, My, Your}
• NOUN	=	{Dog, Car, Computer}
• VERB	=	{Ran, Crashed, Accelerated}
• ADJECTIVE	=	{Purple, Spotted, Old}

Regular expressions

- Second recursive operator: **choice** (sometimes called ”union”, or “or”)

- Two REs can be choiced together using the “|” operator
  - RE1 = “a”, RE2 = “b”
  - The choice is: RE1|2 = “a|b”
  - Matches either

Example:

```
OP   = "*|+"
CMP  = "==|<=|>="
```

Python:

```
import re
re.fullmatch("*|+", "+")
re.fullmatch("==|<=|>=", "==")
```

Can we define these tokens yet?

Tokens		Tokens Definitions
• ARTICLE	=	{The, A, My, Your}
• NOUN	=	{Dog, Car, Computer}
• VERB	=	{Ran, Crashed, Accelerated}
• ADJECTIVE	=	{Purple, Spotted, Old}

Can we define these tokens yet? Yes!

Tokens		Tokens Definitions
• ARTICLE	=	“The|A|Mine|Your”
• NOUN	=	“Dog|Car|Computer”
• VERB	=	“Ran|Crashed|Accelerated”
• ADJECTIVE	=	“Purple|Spotted|Old”

Can we define these tokens yet?

```
ID     = [characters]
NUM    = [numbers]
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = [“ “]
```

Can we define these tokens yet? No!

```
ID     = [characters]
NUM    = [numbers]
ASSIGN = ”=“
PLUS   = “+”
MULT   = “*”
IGNORE = [“ “]
```

Regular expressions

- Last recursive operator: **Repeat**

- Unary operator: *
  - RE1 = “a”
  - Repeat RE1 zero or more times: ”a*”

Example:

```
RE1   = “a*“
RE2   = “a*|b*”
RE3   = ”a|b*
```

Python:

```
import re
re.fullmatch(“a*|b*“, “aaa“)
re.fullmatch(“a*|b*”, “”)
```

Regular expressions

- Last recursive operator: **Repeat**

- Unary operator: *
  - RE1 = “a”
  - Repeat RE1 zero or more times: ”a*”

Example:

```
RE1   = “a*“
RE2   = “a*|b*”
RE3   = ”a|b*
```

Precedence?

Python:

```
import re
re.fullmatch(“a*|b*“, “aaa“)
re.fullmatch(“a*|b*”, ““)
```

Regular expressions

- These are the theoretical foundational operators.
- Most languages give syntactic sugar to make common cases easier
- Most languages also break the theory
  - Perl regexes are extremely complicated
    - https://www.perlmonks.org/?node_id=809842
  - Python regexes (with recursion) are can capture context free languages
    - https://www.npopov.com/2012/06/15/The-true-power-of-regular-expressions.html#matching-context-free-languages

Regular expressions

- strict repeat operator: +
- one or more repeats (the * operator is 0 or more repeats)
- derivation: “r+” = “rr*”

Regular expressions

- Ranges:
  - digits [0-9]
  - alpha [a-z], [A-Z]
- Derivation: [0-9] = ”1|2|3|4|5|6|7|8|9|0”
- Lets try C style IDs: [a-zA-Z_][a-zA-Z_0-9]*
- Hexadecimal numbers:

Regular expressions

- Ranges:
  - digits [0-9]
  - alpha [a-z], [A-Z]
- Derivation: [0-9] = ”1|2|3|4|5|6|7|8|9”
- Lets try C style IDs: “[a-zA-Z][0-9a-zA-Z]*”
- Hexadecimal numbers: “0x[0-9a-fA-F]+”

Regular expressions

- optional operator ?
  - optional characters

- “r?” = “|r”

- Example: “ab?”

Regular expressions

- optional operator ?
  - optional characters
- “r?” = “|r”
- Example: “ab?”
- Let’s do simple floating-point numbers:

Regular expressions

- optional operator ?
  - optional characters

- “r?” = “|r”

- Example: “ab?”

- Let’s do simple floating-point numbers: “[0-9]+(\.[0-9]+)?”

Regular expressions

- any character ‘.’

- example using email (this is probably too general!)

Regular expressions

- any character ‘.’
- example using email (this is probably too general!)
- ”.*@.*\.com”

Using REs

- What if we want to extract the domain or user name from the email?
- We can use groups!
  - use ()s to delimitate groups
- ”(.*)@(.*\.com)”
- Index the resulting object with [1] and [2] to get to the user name and domain respectively

Using REs

- you can give groups id names rather than using indices

- “(?P<name>.+)@(?P<domain>.+\.com)”

Example Using RE groups

- you can give groups id names rather than using indices

  “(?P<name>.+)@(?P<domain>.+\.com)”

```
import re
pattern = r"(?P<name>.+)@(?P<domain>.+\.com)"
email = "johndoe@example.com"
match = re.match(pattern, email) # apply pattern
if match:
    name = match.group("name") # extract user name
    domain = match.group("domain") # extract domain
    print(f"Name: {name}, Domain: {domain}")
else:
    print("No match found.")
```

REs are good for?

- Scanning large amounts of documents quickly, looking for:
  - Websites
  - Email
  - Profiling numbers
  - Variable usages
  - **What else?**

RE examples

- **What can REs not do?**

- Nested structures, such as parenthesis matching:
  - Try doing arithmetic expressions
  - You will not be able to match ()s

- Classical example: REs cannot capture same number of repeats:
  - A{N}B{N}

- REs cannot parse HTML!!!
  - One of the most upvoted answers on stackoverflow!
  - https://stackoverflow.com/questions/1732348/regex-match-open-tags-except-xhtml-self-contained-tags/1732454#1732454

For your homework

- You'll need to write tokens for a simple programming language, including:

```
ID     = [characters]
NUM    = [numbers]
ASSIGN = "="
PLUS   = "+"
MULT   = "*"
IGNORE = [" "]
```

How to implement an RE matcher?

- Overview: first you have to parse the RE...
  - Chicken and egg problem
  - The language of REs is not a regular language. It is context sensitive (because it has ()s)
  - But once you can parse the RE, there are several options

How to implement an RE matcher?

- Convert to an automata
  - Learn more about this CSE103
  - A cool website
  - https://ivanzuzak.info/noam/webapps/fsm_simulator/

How to use REs in a scanner implementation?

- Will be the next topic of discussion.