REs, FSMs, Forth, and CFGs

Part 2 of 3

Three things today

The foundations of regular expressions

(Don’t need to remember details)

Introduction to grammars

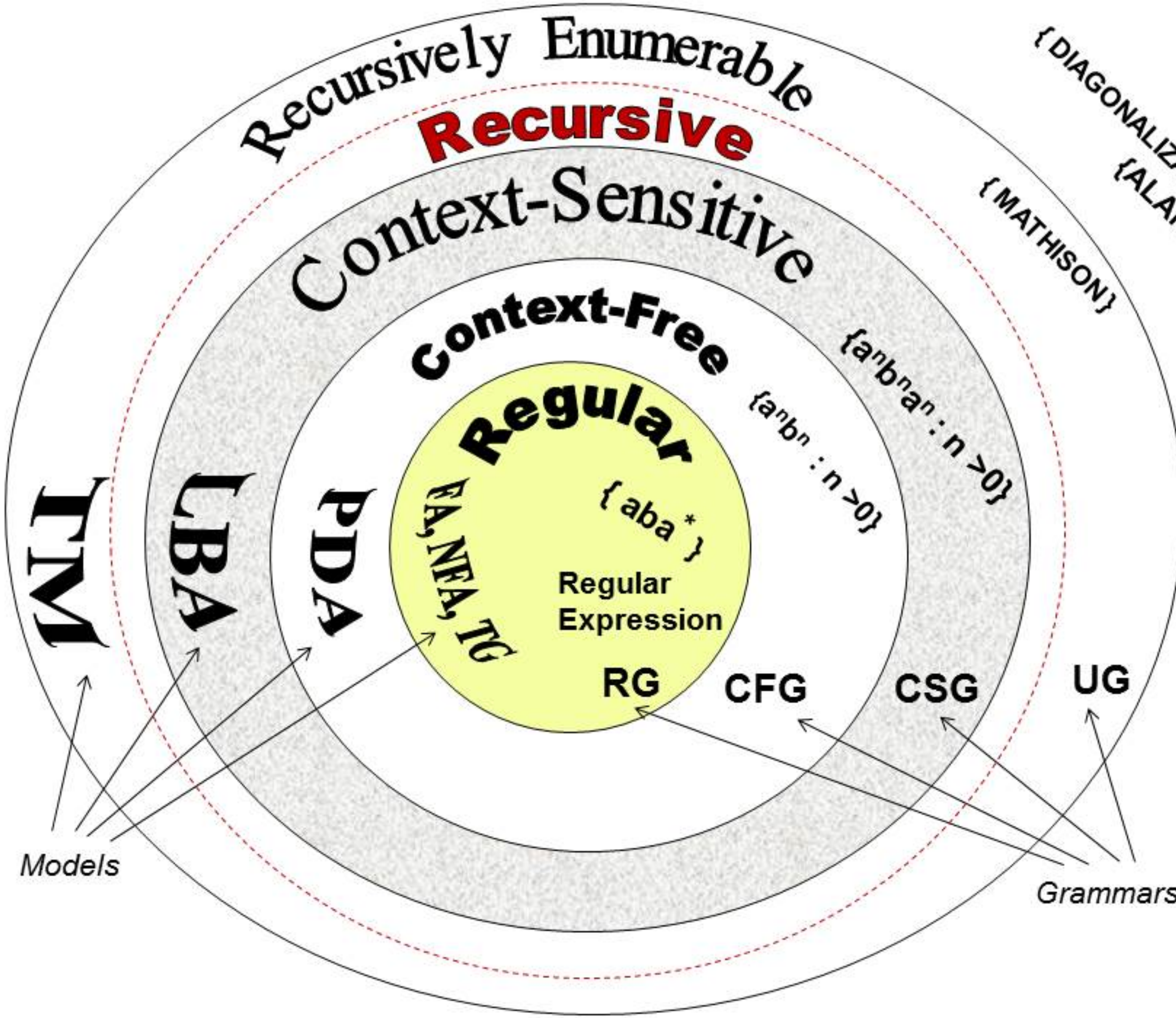
(Important to get concepts)

Intro to FORTH

(You’ll need this for the lab)

Regular expressions have a *nice property*…

If you give me a regex and a string, I can check if that string matches the regex in **linear time**



Recursively Enumerable
Recursive
Context-Sensitive
Context-Free
Regular
{ DIAGONALIZATION }
{ALAN }
{ MATHISON}
{a^n b^n a^n : n >0}
{a^n b^n : n >0}
{ aba * }
Regular Expression
TM
LBA
PDA
FA, NFA, TG
RG
CFG
CSG
UG
Models
Grammars

Can I cook up a regular expression that will classify any string?

(No…)

If I could, it would imply I could solve any problem in linear time!

So what’s an example of a regular expression I couldn’t write?

“The set of strings P such that P…?”

So what's an example of a regular expression I couldn't write?

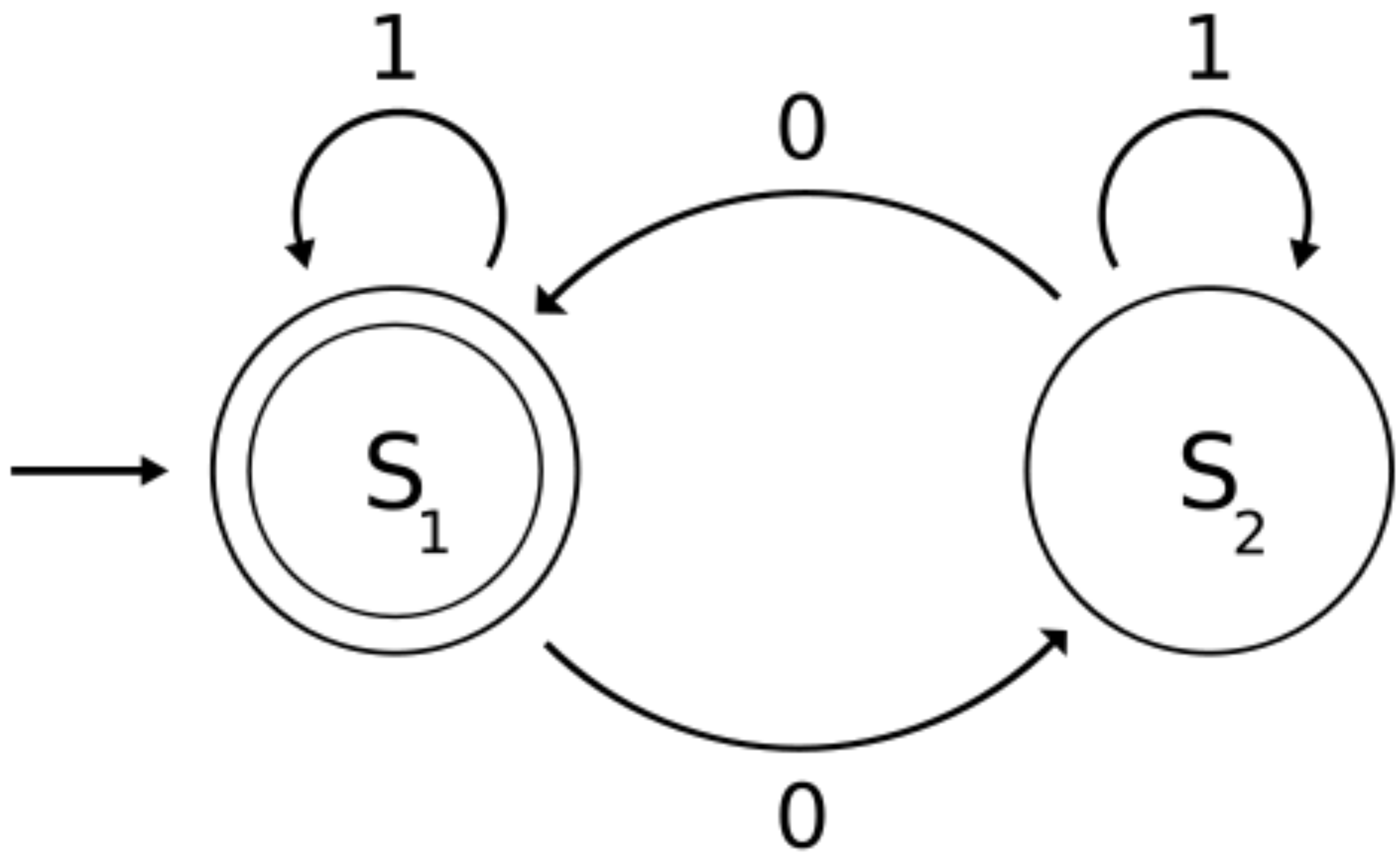
"The set of strings P such that P…?"

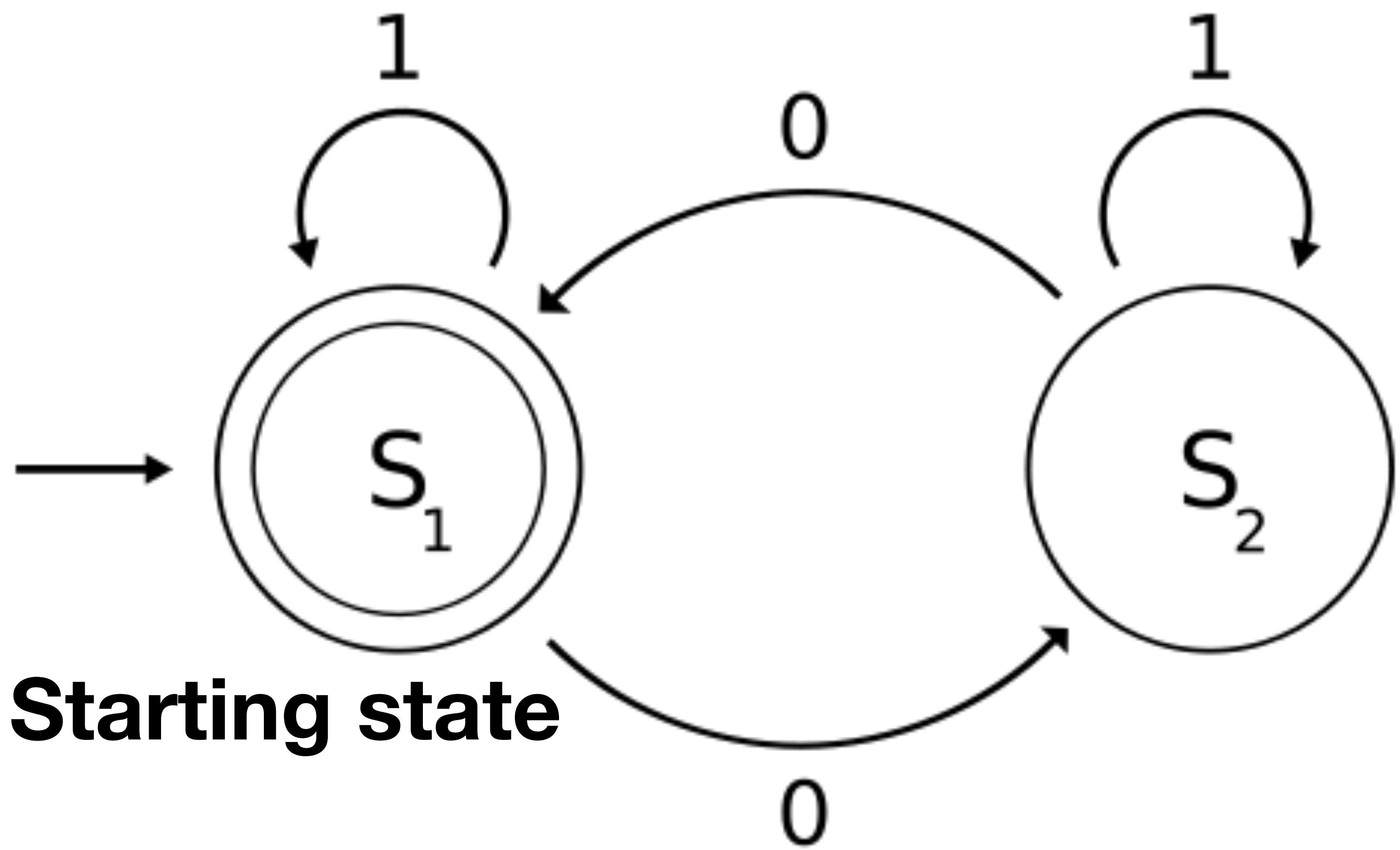
(**Answer**: is a program that halts)

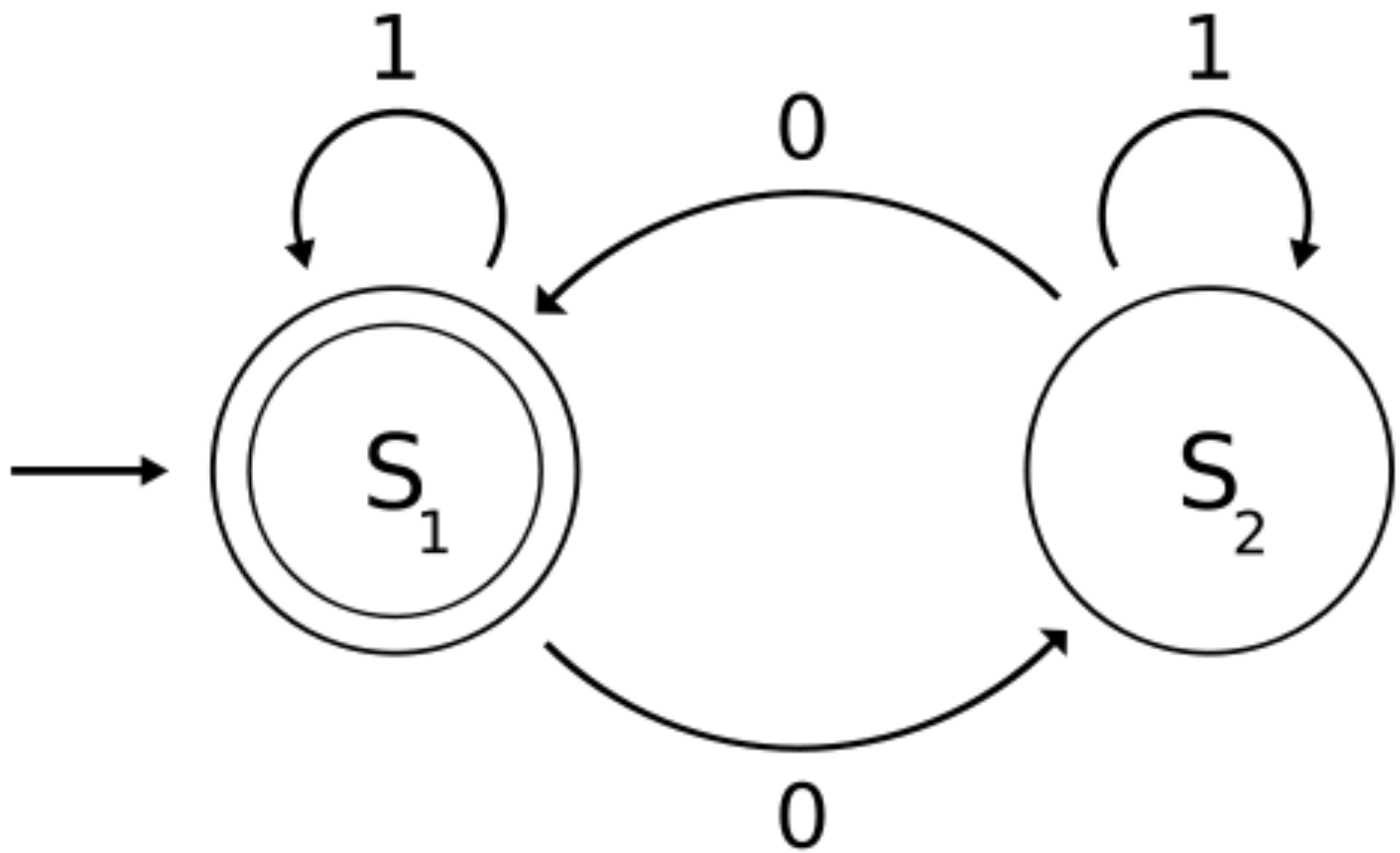
Regular expressions can be **implemented** using **finite state machines**

We won’t talk too much about FSMs in this class

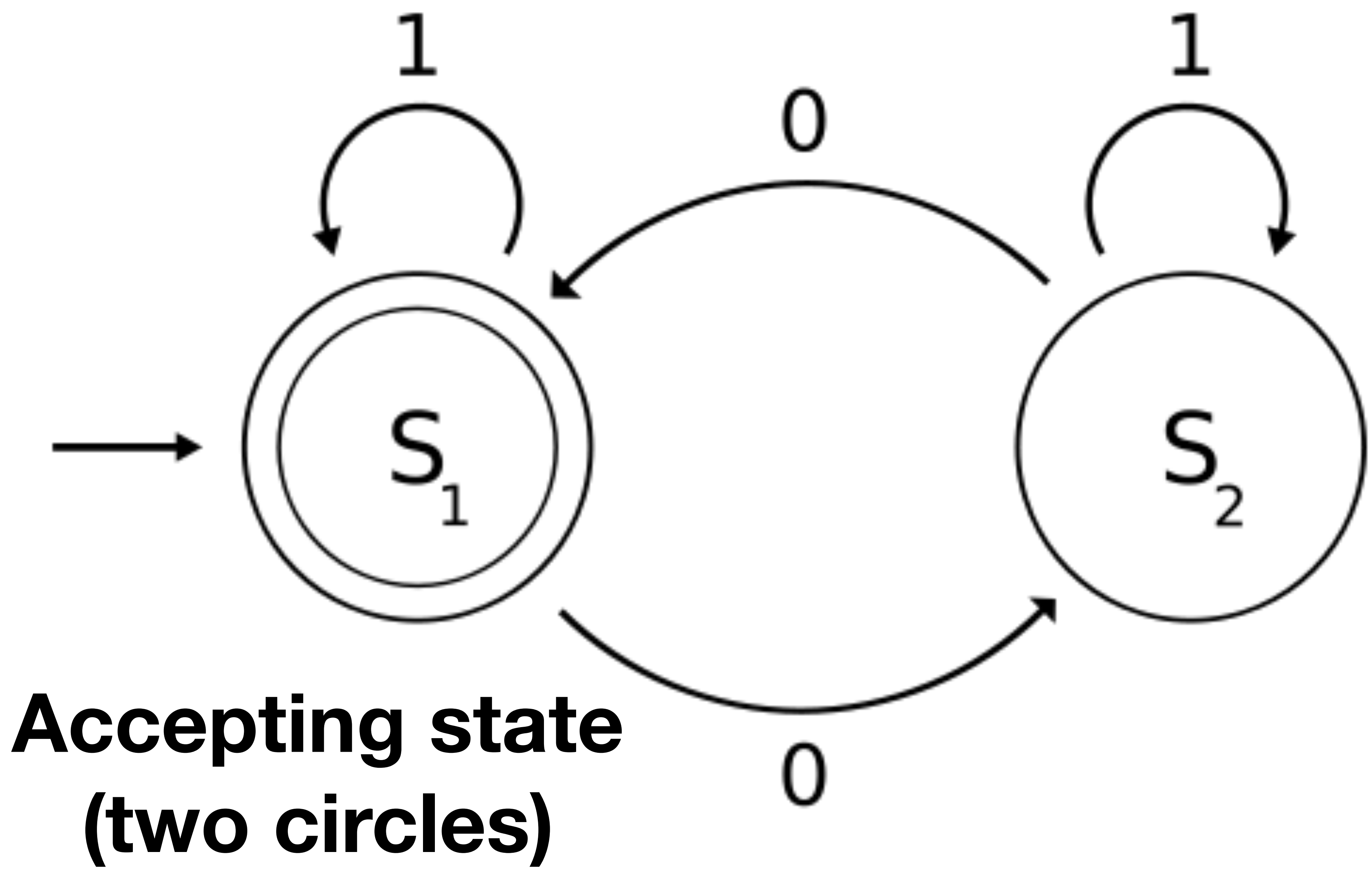
All regexes can “compile” (turn to, in systematic way) FSM


1
0
1
S_1
S_2
0

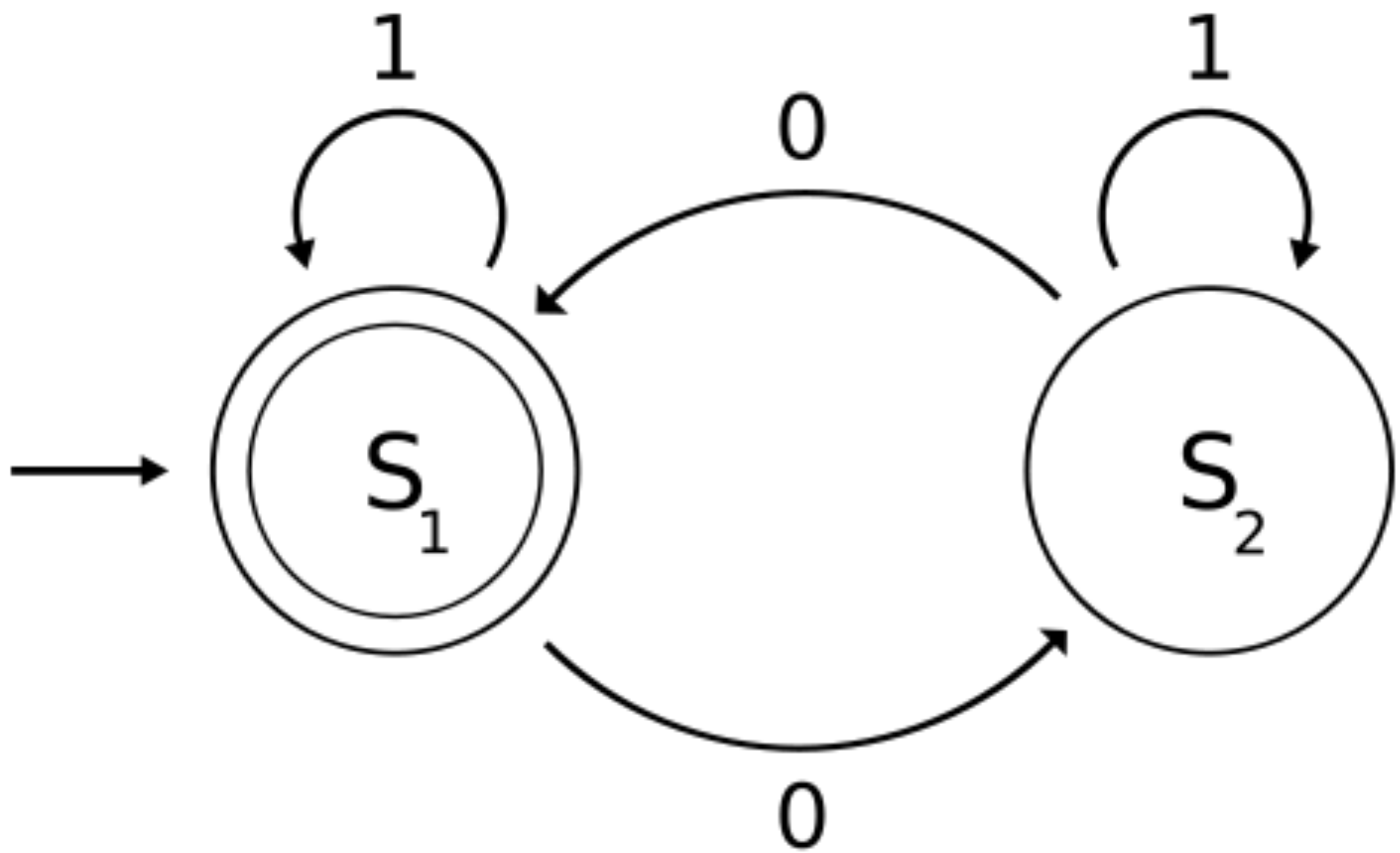

1
0
1
S_1
S_2
Starting state
0



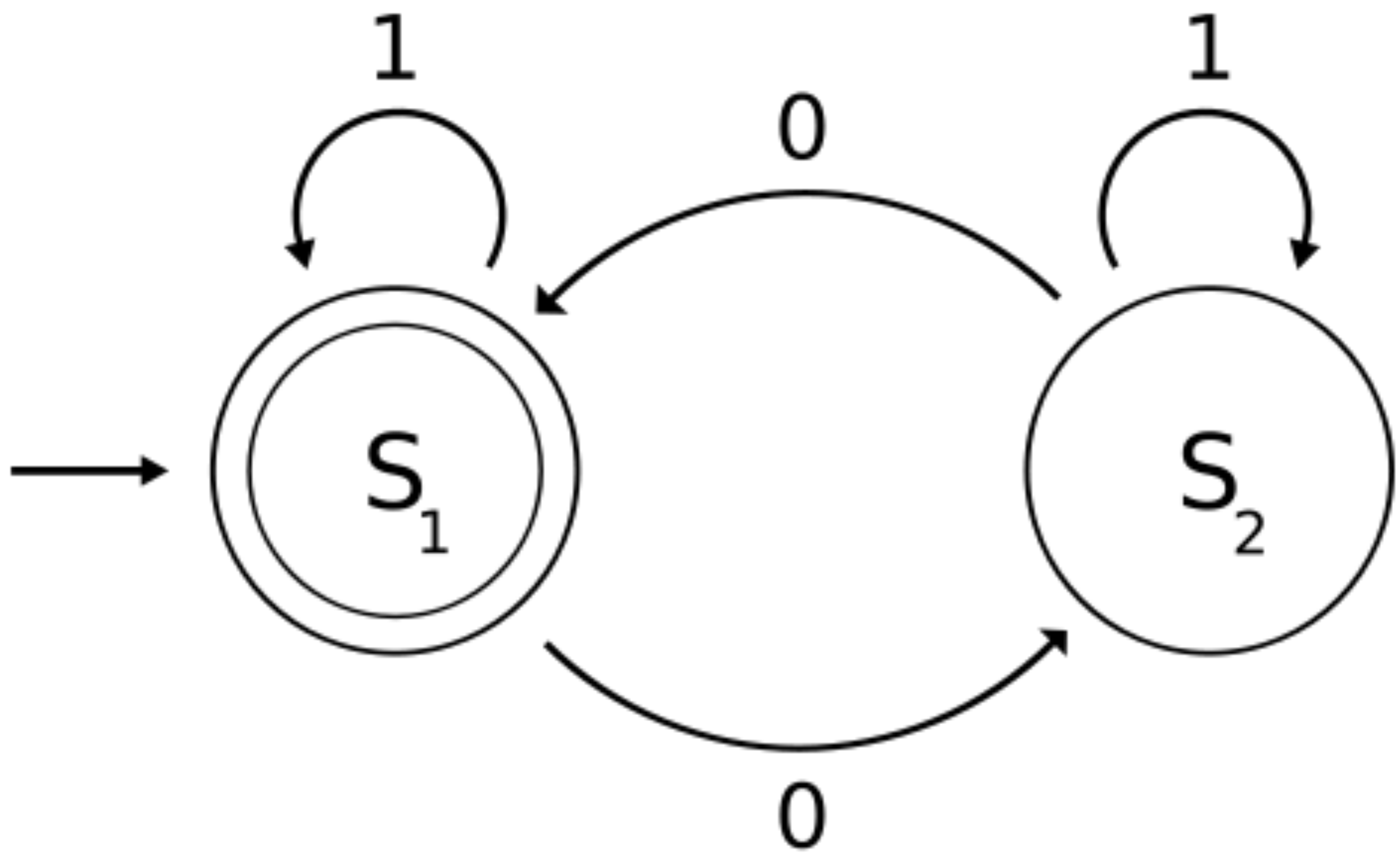
Transition on input


1
0
1
S_1
S_2
0
Accepting state
(two circles)

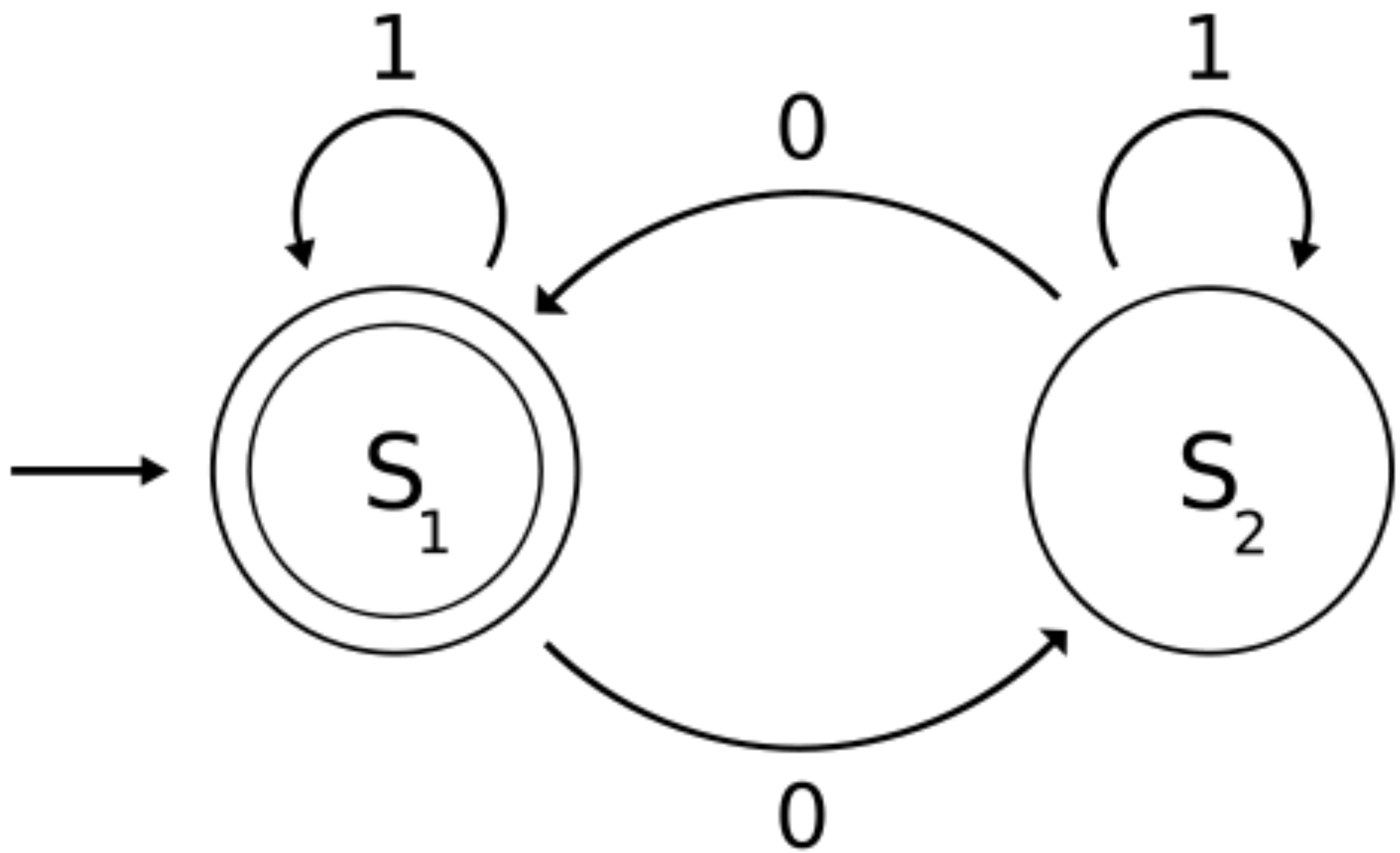
011 S1



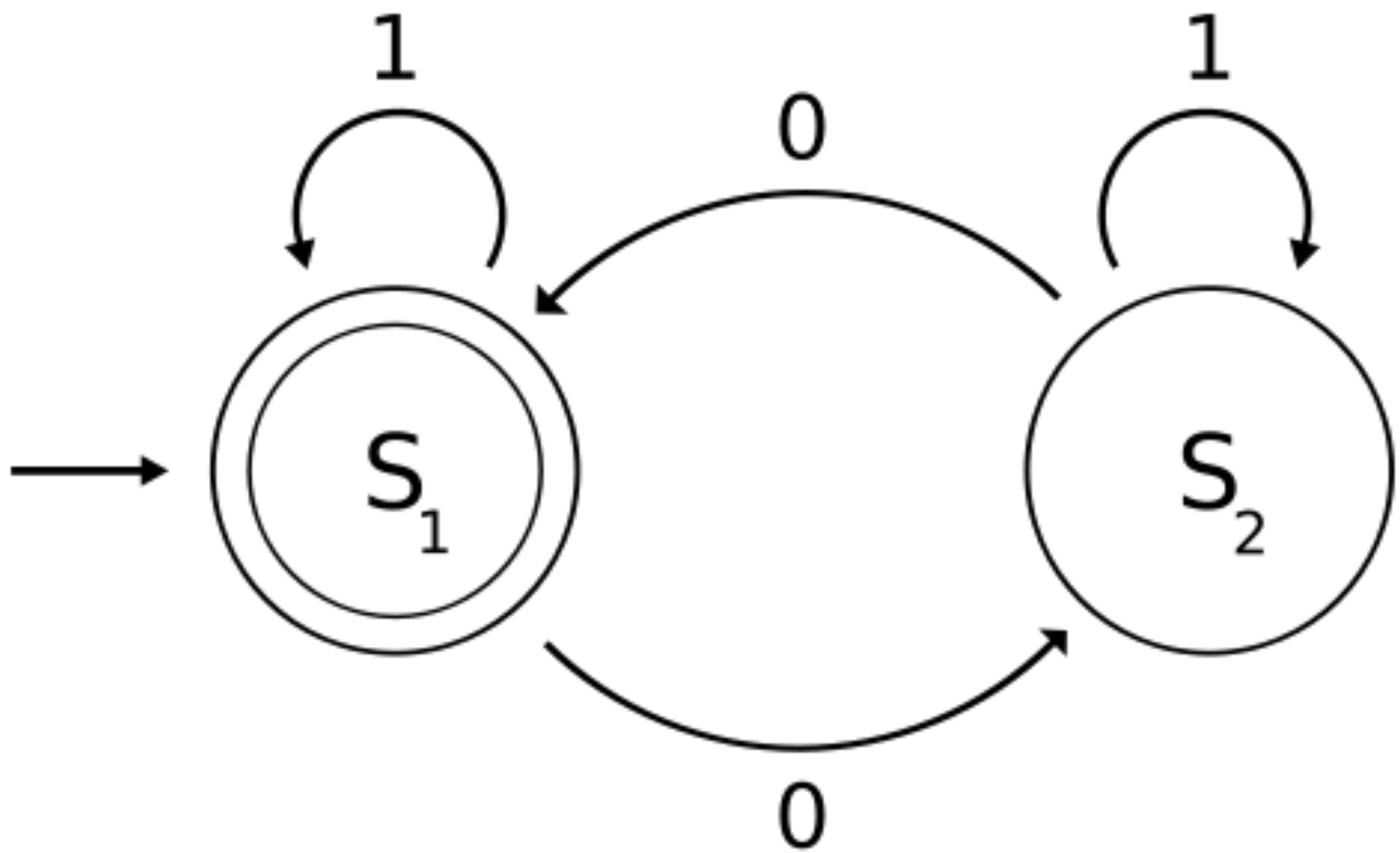
011 S2



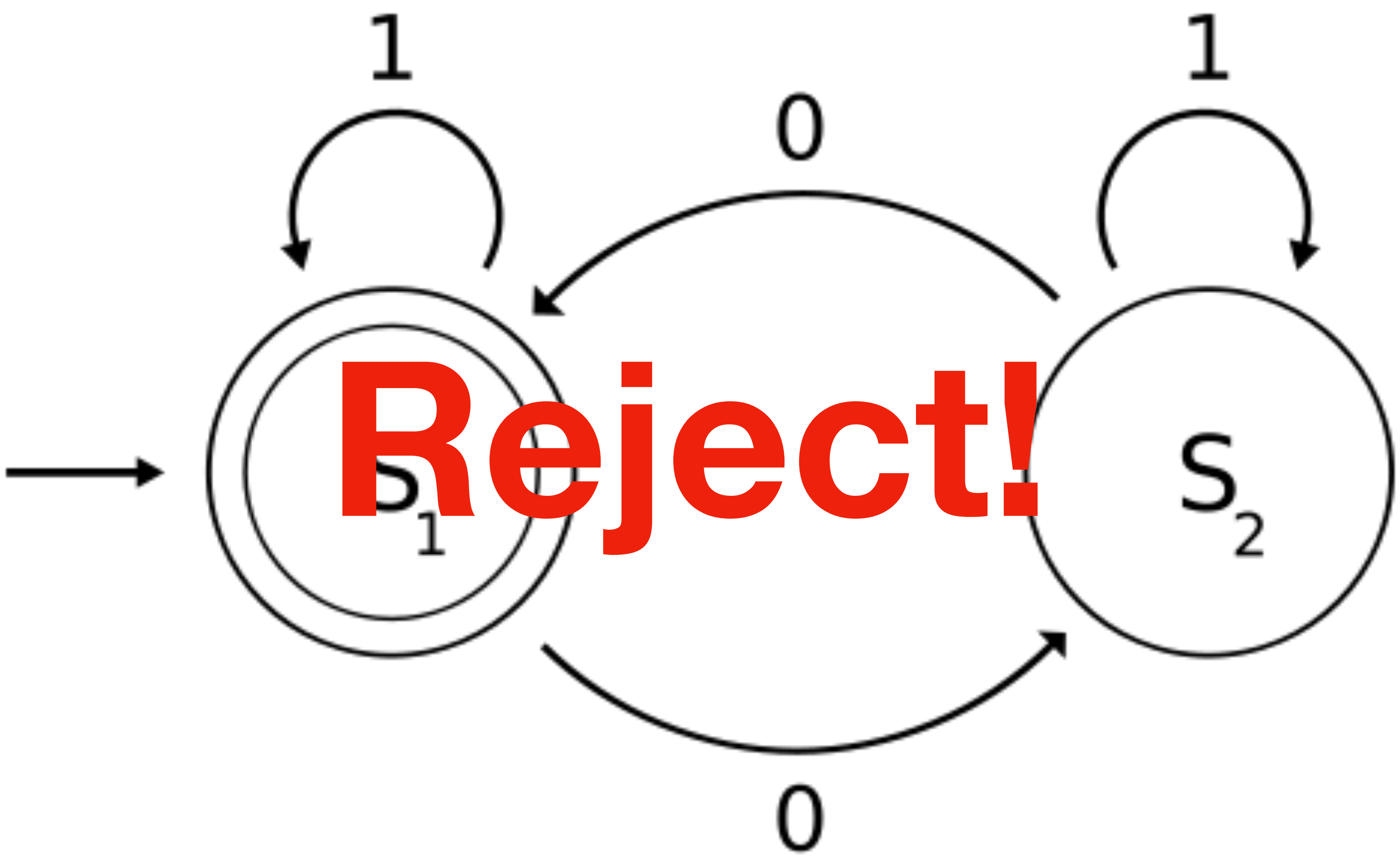
<u>01</u>1 S2 **Stay!**



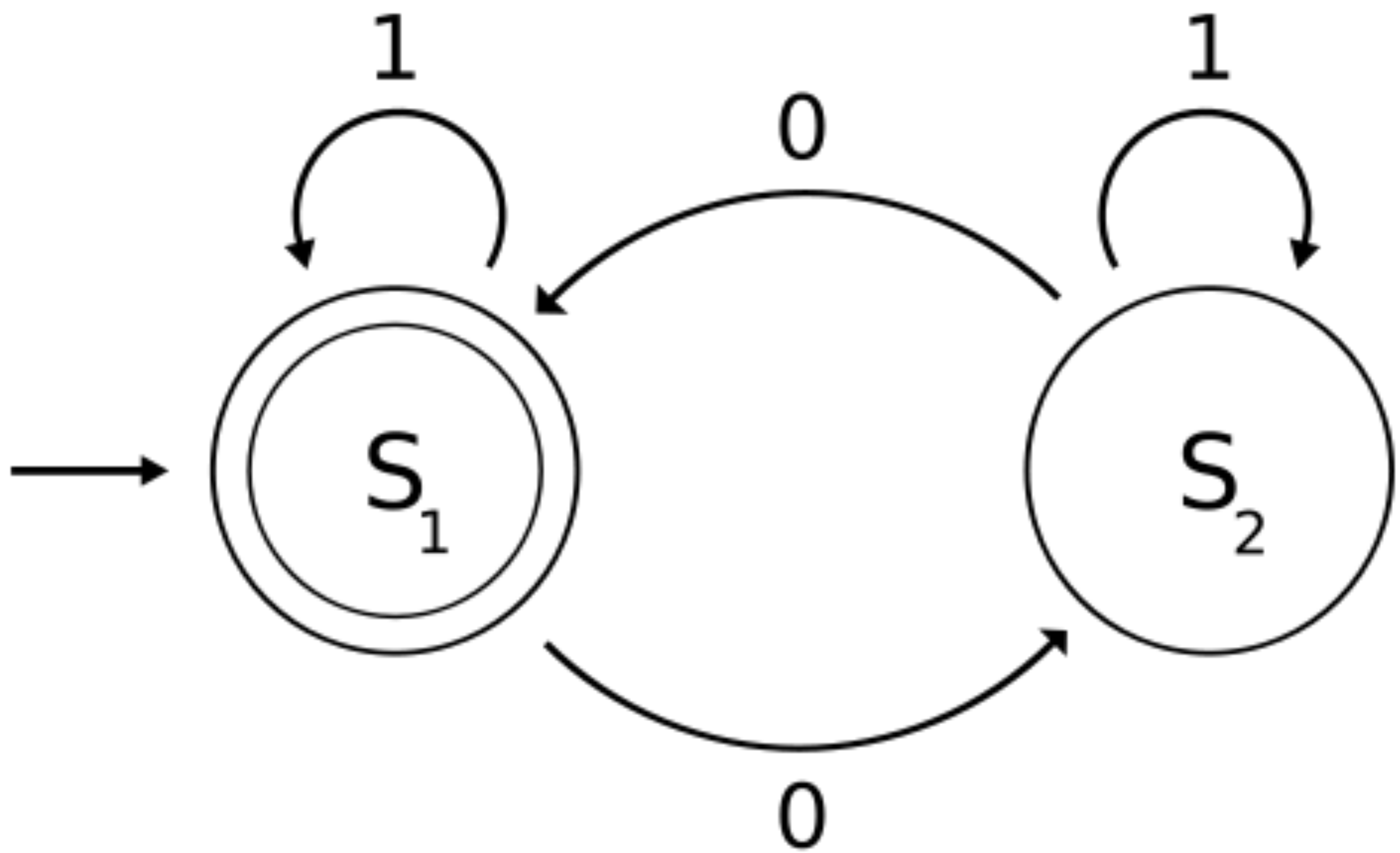
<u>011</u> S2



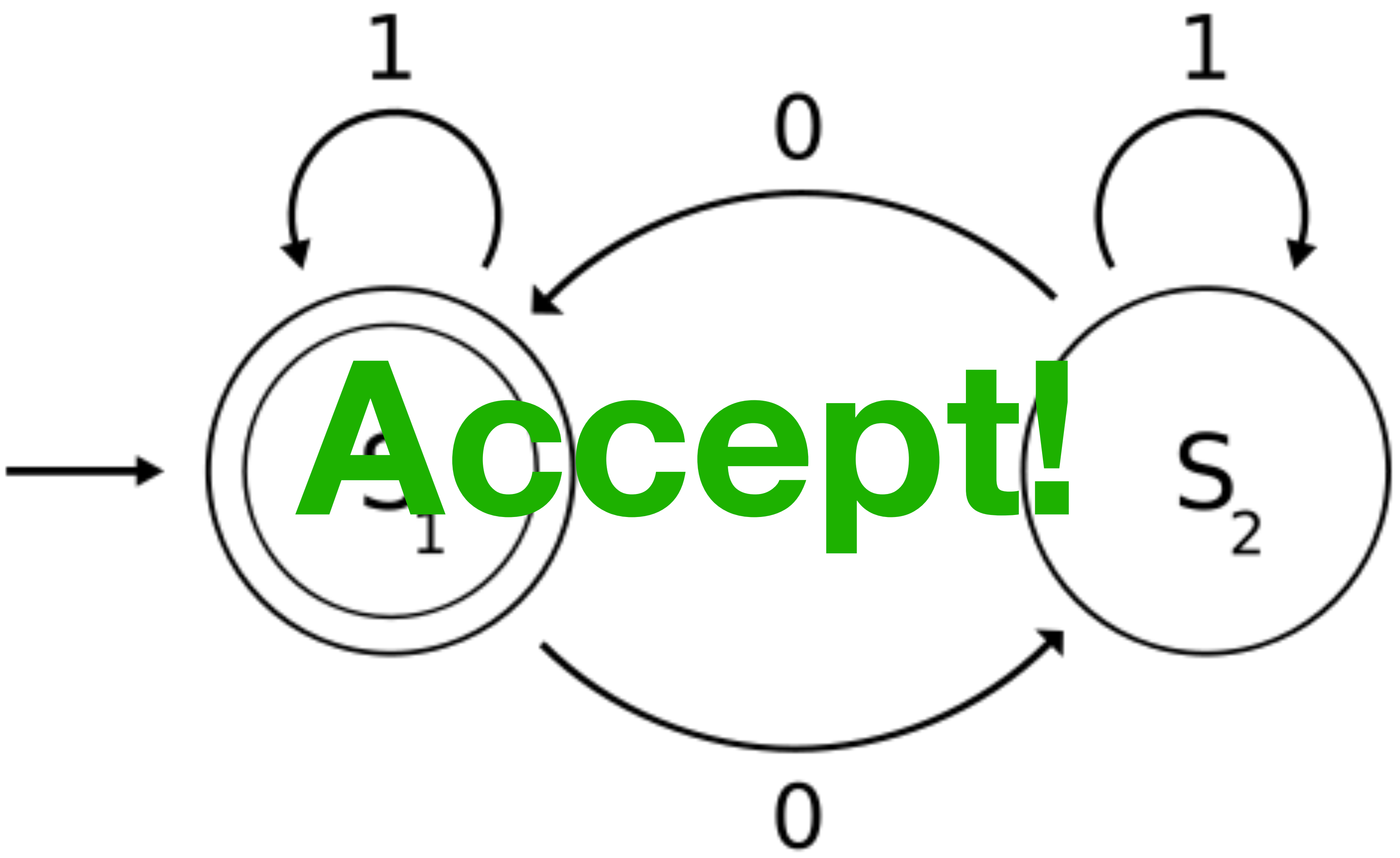
<u>011</u> S2



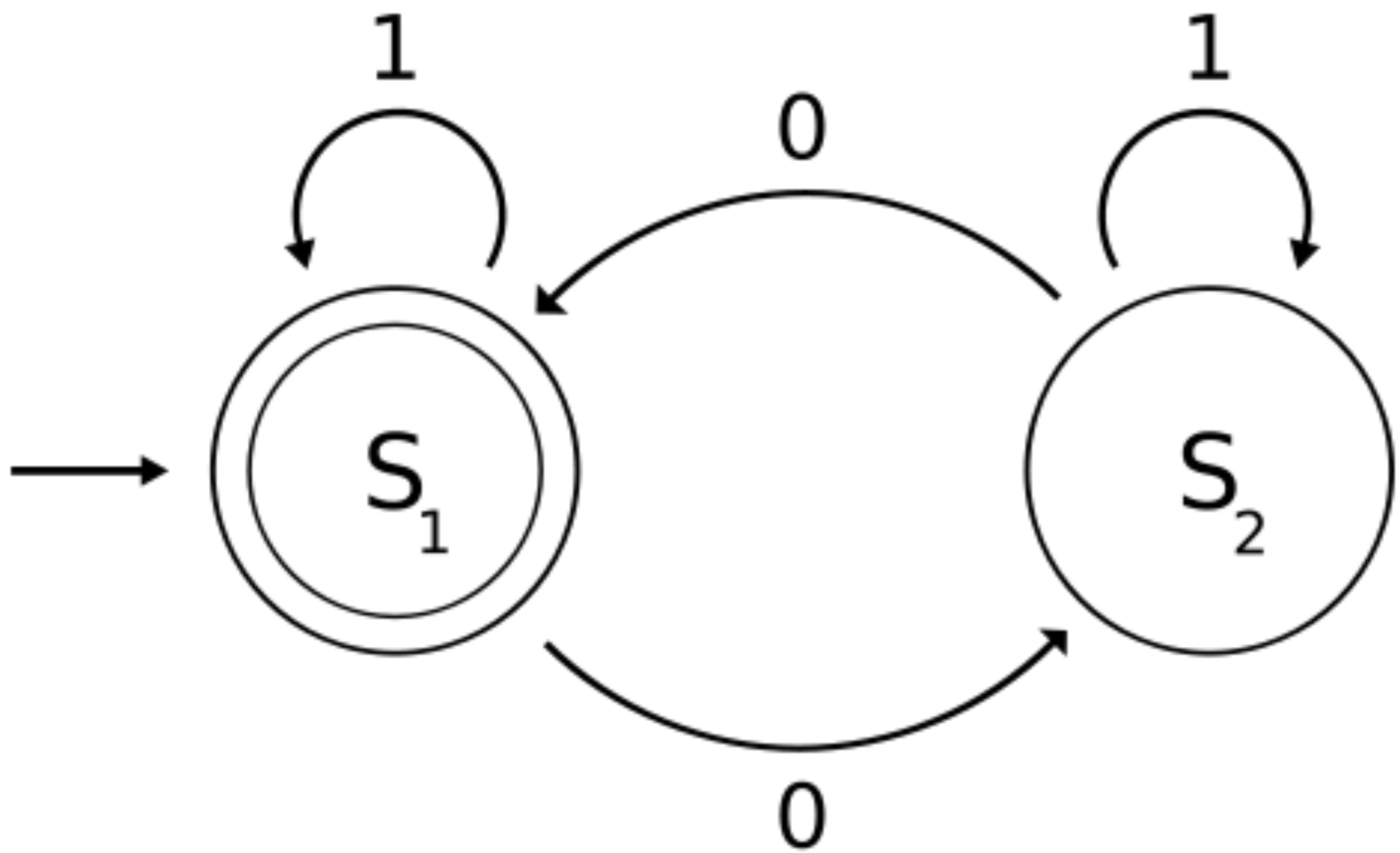
<u>0110</u> S1



<u>0110</u> S1

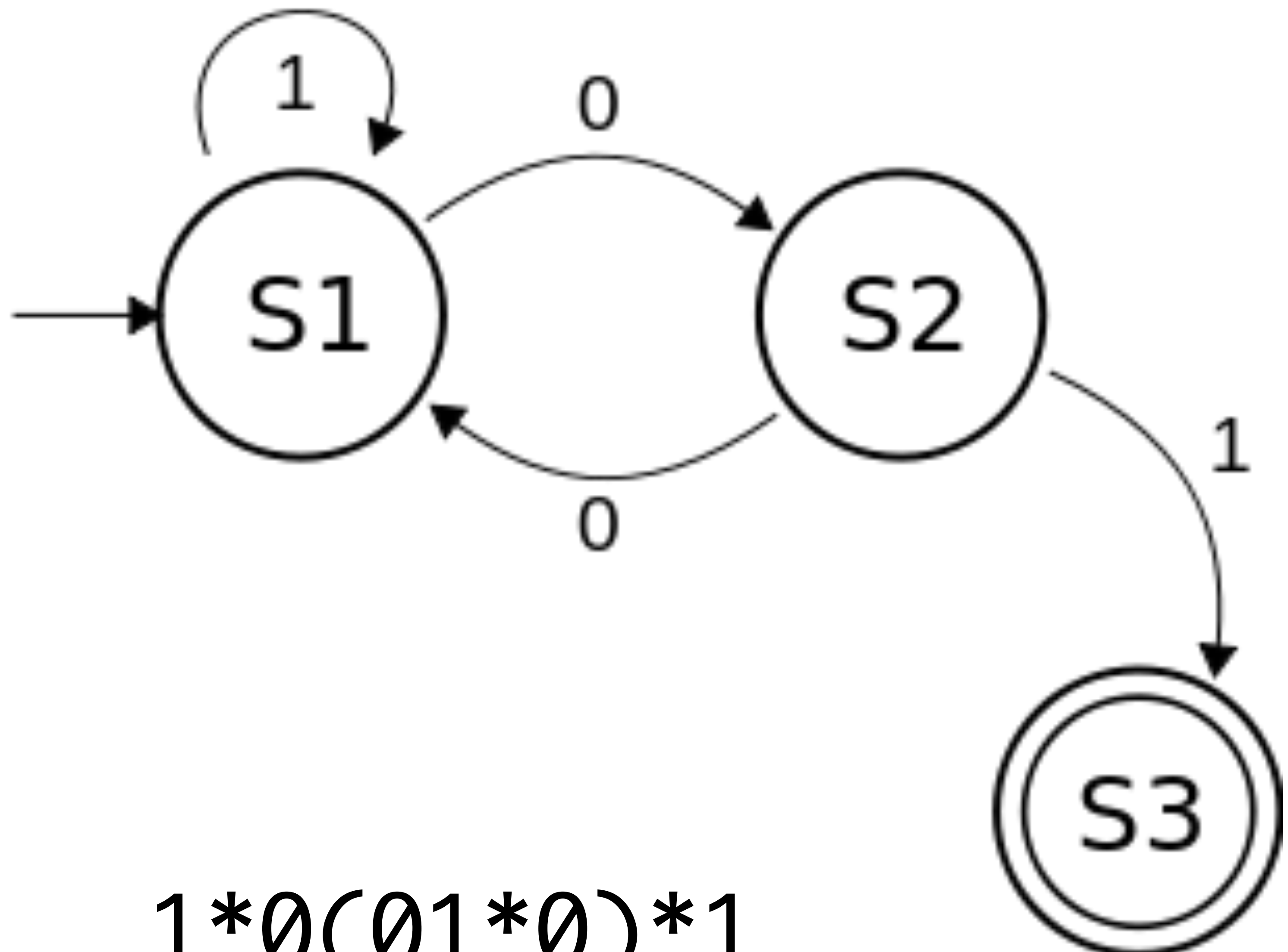


(1|01*0)*



Note that I got this wrong in class

“Any number of 1s, followed by an even number of 0s, followed by a single 1”



1*0(01*0)*1

Note that I got this wrong in class

Idea: FSMs remember only "one state" of memory

It's kind of like programming with only one register (of unbounded width)

Theorem: for every regex, a corresponding FSM exists, and vice versa

Q: Why is this useful?

Theoretical A: Bedrock automata theory, useful in proving computational bounds

Practical A: Efficient regex implementation

Motivating CFGs

Parenthesis are **balanced** when each left matches a right

```
{}

{{}}

{{{}}}

{{{{}}}}
```

Balancing parentheses necessary to check program syntax
(e.g., for C++)

{*}* doesn’t work

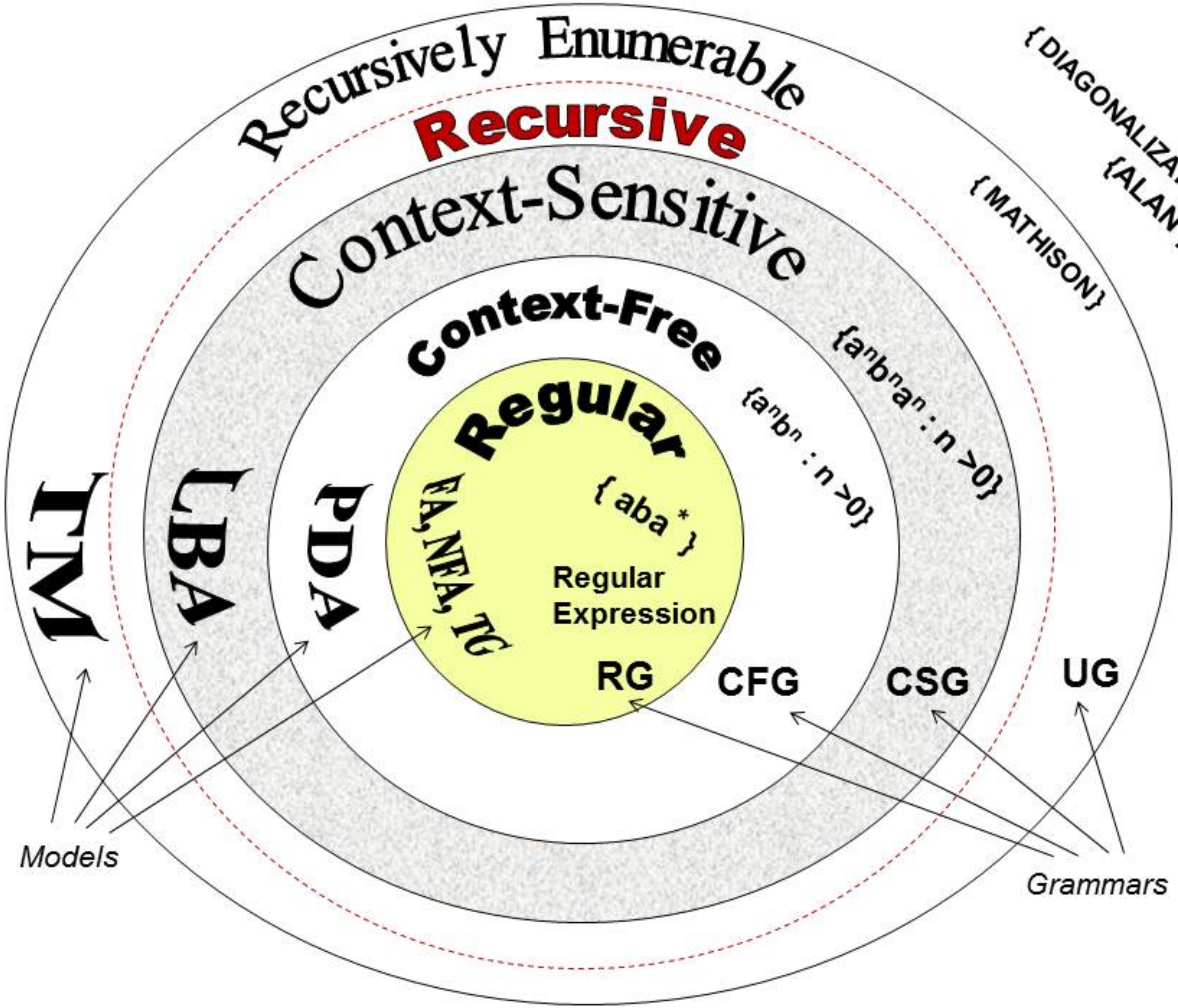
Turns out: it is **impossible** to write a regex to capture this fact

Instead, we will use *context-free grammars*

Here’s a grammar that matches balanced parentheses

```
S -> ε
S -> { S }
```

We’ll talk more about grammars later today and on Friday



Recursively Enumerable
Recursive
Context-Sensitive
Context-Free
Regular
{ DIAGONALIZATION }
{ALAN }
{ MATHISON}
{$a^nb^na^n$: n >0}
{a^nb^n : n >0}
{ aba * }
Regular Expression
TM
LBA
PDA
FA, NFA, TG
RG
CFG
CSG
UG
Models
Grammars

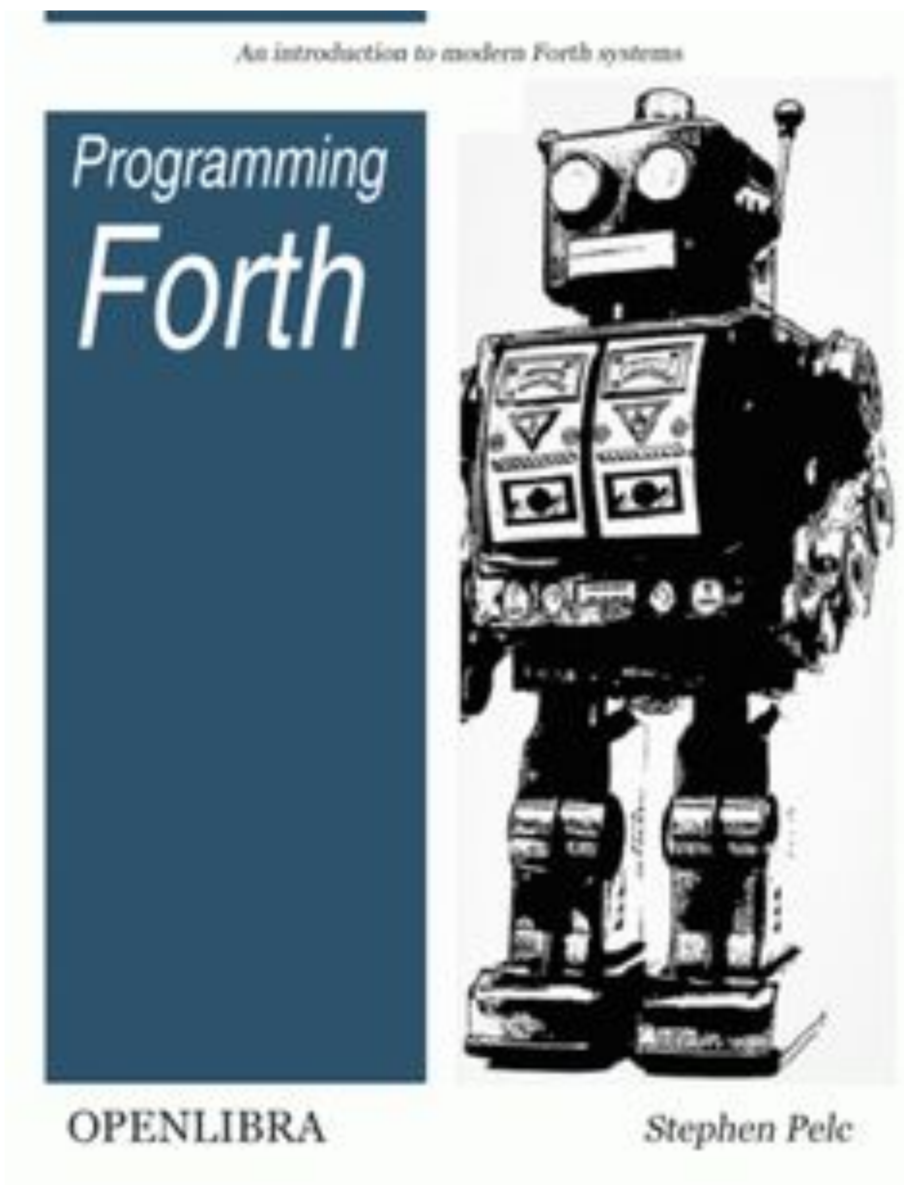
CFG’s are **more expressive** than regular expressions, and commensurately more **complex** to check

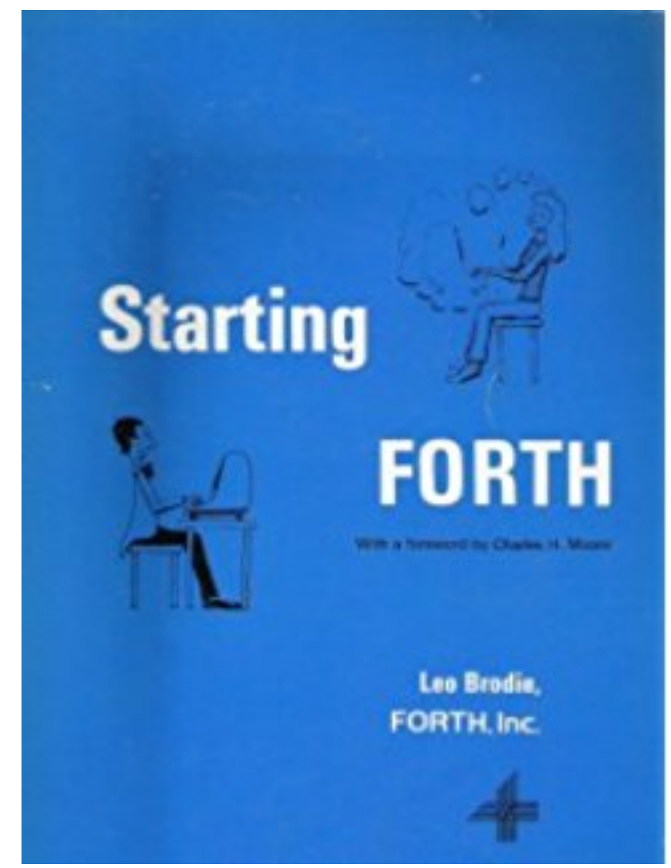
Whereas regular expressions are modeled by finite state machines, CFGs are modeled by state machines that also can push / pop a **stack**

But what programming languages can we implement **right now**

(Without needing to implement CFGs)

FORTH


An introduction to modern Forth systems
Programming
Forth
OPENLIBRA
Stephen Pelc


Starting
FORTH
Leo Brodie,
FORTH, Inc.

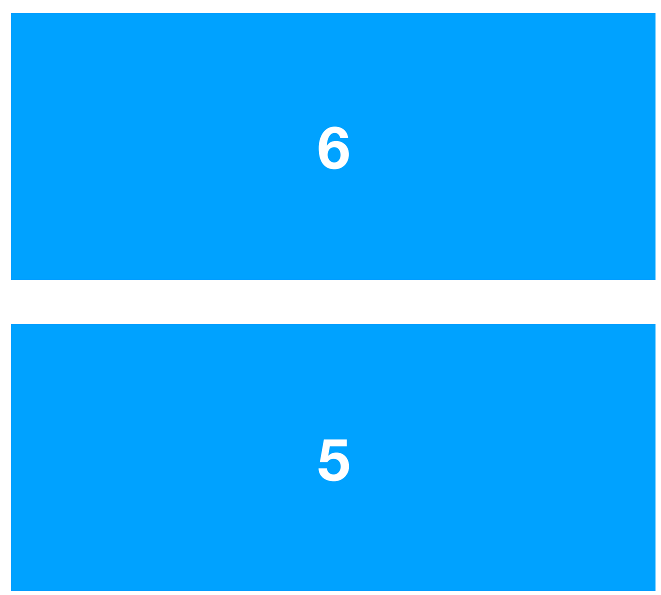
Forth is a **stack-based** language

A beginner’s guide to FORTH

http://galileo.phys.virginia.edu/classes/551.jvn.fall01/primer.htm

Assembly uses registers and memory, but FORTH uses a stack as its main abstraction

5


6
5

+

6

5

+

11

You have **already implemented** parts of forth

Each command in forth is called a **word**

Words manipulate the stack

$(\; x_1 \; -- \;)$

drop

Drops the most recent thing on the stack

swap

$(\; x_1 \; x_2 \; \text{--} \; x_2 \; x_1 \;)$

$\uparrow$

Top!

nip

$(\; x_1 \; x_2 \; \text{--} \; x_2 \;)$

dup

$(\ x_1 \ \text{--} \ x_1 \ x_1)$

over

$(\; x_1 \; x_2 \; \text{--} \; x_1 \; x_2 \; x_1 \;)$

tuck

$(\; x_1 \; x_2 \; -- \; x_2 \; x_1 \; x_2 \;)$

You can define **your own** words (functions)

```
: add1 1 + ;
```

Adding two Euclidian points

```
x1 y1 x2 y2 -> (x1 + x2) (y1 + y2)
```

Want to define **addcartesian** word, which does this:

```
1 2 3 4  ok
addcartesian  ok
.s <2> 4 6  ok
```

Adding two Euclidian points

```
x1 y1 x2 y2 -> (x1 + x2) (y1 + y2)
```

rot

```
x1 y1 x2 y2 -> x1 x2 y2 y1
```

+

```
x1 x2 y2 y1 -> x1 x2 (y1+y2)
```

What do I do from here?

Adding two Euclidian points

```
x1 y1 x2 y2 -> (x1 + x2) (y1 + y2)
```

rot

```
x1 y1 x2 y2 -> x1 x2 y2 y1
```

+

```
x1 x2 y2 y1 -> x1 x2 (y1+y2)
```

rot

```
x1 x2 (y1+y2) -> x2 (y1+y2) x1
```

rot

```
x2 (y1+y2) x1 -> (y1+y2) x1 x2
```

+

```
(y1+y2) x1 x2 -> (y1+y2) (x1+x2)
```

swap

```
(y1+y2) (x1+x2) -> (x1+x2) (y1+y2)
```

So that’s forth, we’ll touch a bit more of it Friday

And you’ll be implementing part of it in Lab 4

Back to CFGs!

Why? Because most languages use infix operators

Here's a context free grammar

```
Expr -> number
Expr -> Expr + Expr
Expr -> Expr * Expr
```

Formally, a grammar is…

- A set of **terminals**
  - These are the things you **can't rewrite any further**
- A set of **nonterminals**
  - These are the things you **can rewrite further**
- A set of **production rules**
  - These are a bunch of **rewrite rules**
- A **start symbol**

Terminals = {number, +, *}

Nonterminals = {Expr}

Productions =

```
Expr -> number
Expr -> Expr + Expr
Expr -> Expr * Expr
```

Start symbol = Expr

To determine if a grammar matches an expression, **you play a game**

1 + 2

```
Expr -> number
Expr -> Expr + Expr
Expr -> Expr * Expr
```

First, start with a nonterminal and write that on the page

1 + 2

```
Expr -> number
Expr -> Expr + Expr
Expr -> Expr * Expr
```

First, start with a nonterminal and write that on the page

```
Expr
```

1 + 2

```
Expr -> number
Expr -> Expr + Expr
Expr -> Expr * Expr
```

First, start with a nonterminal and write that on the page

```
Expr
```

To play the game: attempt to apply each production so that you arrive at your full expression

1 + 2

```
Expr -> number
Expr -> Expr + Expr
Expr -> Expr * Expr
```

First, start with a nonterminal and write that on the page

```
Expr -> Expr + Expr
```

1 + 2

```
Expr -> number
Expr -> Expr + Expr
Expr -> Expr * Expr
```

First, start with a nonterminal and write that on the page

```
Expr
-> Expr + Expr
-> number + Expr
-> number + number
-> 1 + number
-> 1 + 2
```

1 + 2

```
Expr -> number
Expr -> Expr + Expr
Expr -> Expr * Expr
```

First, start with a nonterminal and write that on the page

Some moves don’t lead you to winning the game.

1 + 2

```
Expr -> number
Expr -> Expr + Expr
Expr -> Expr * Expr
```

First, start with a nonterminal and write that on the page

Some moves don’t lead you to winning the game.

```
Expr
-> Expr * Expr
???
```

```
Expr -> number
Expr -> Expr + Expr
Expr -> Expr * Expr
```

This grammar is **ambiguous**

```
1 + 2 * 3
```

```
Expr
-> Expr + Expr
```

```
Expr
-> Expr * Expr
```

Exercise: complete the derivations from here

We’ll define this more rigorously on Friday

```
Expr -> number
Expr -> Expr + Expr
Expr -> Expr * Expr
```

```
1 + 2 * 3
```

```
Expr
-> Expr + Expr
-> Expr + Expr * Expr
-> number + Expr * Expr
-> number + number * Expr
-> number + number * number
```

```
Expr
-> Expr * Expr
-> Expr + Expr * Expr
-> number + Expr * Expr
-> number + number * Expr
-> number + number * number
```

Famous example from C, the “dangling else”

```
if …
   if …
   else …
```

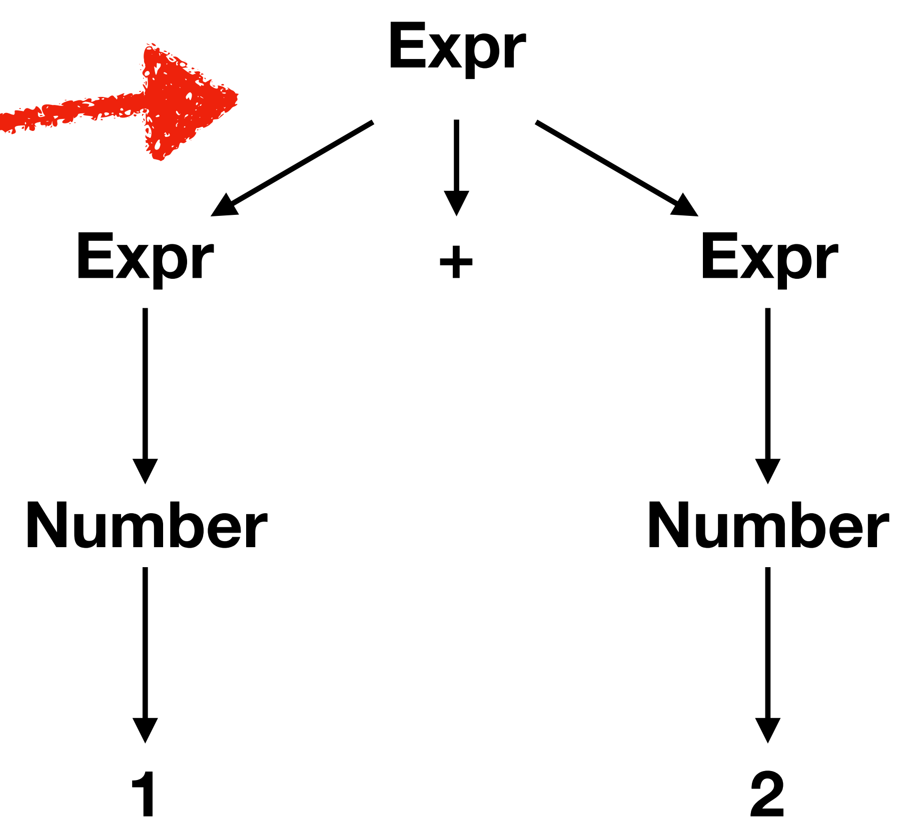
Does the else belong to the first if? Or the second?

(Ans: in C, the second)

Most real languages handle these in hacky one-off ways

We can turn a derivation into a **parse tree**

```
Expr
-> Expr + Expr
-> number + Expr
-> number + number
-> 1 + number
-> 1 + 2
```



This parse tree is a **hierarchical representation** of the data

A **parser** is a program that automatically generates a parse tree

A parser will generate an **abstract syntax tree** for the language

Parsing is **hard**

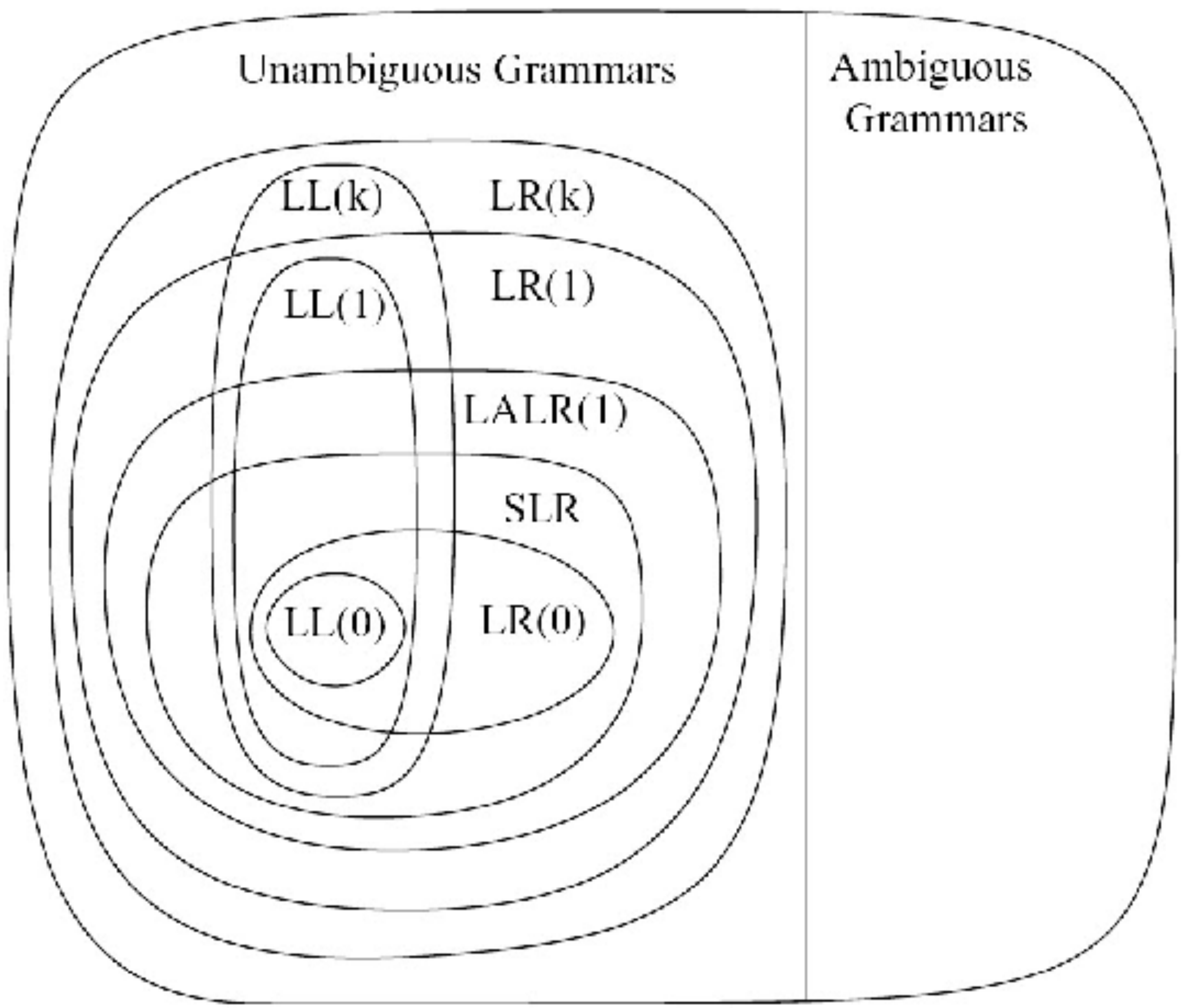
And also **boring**

But an **important problem**

And there are a **ton** of different parsing algorithms

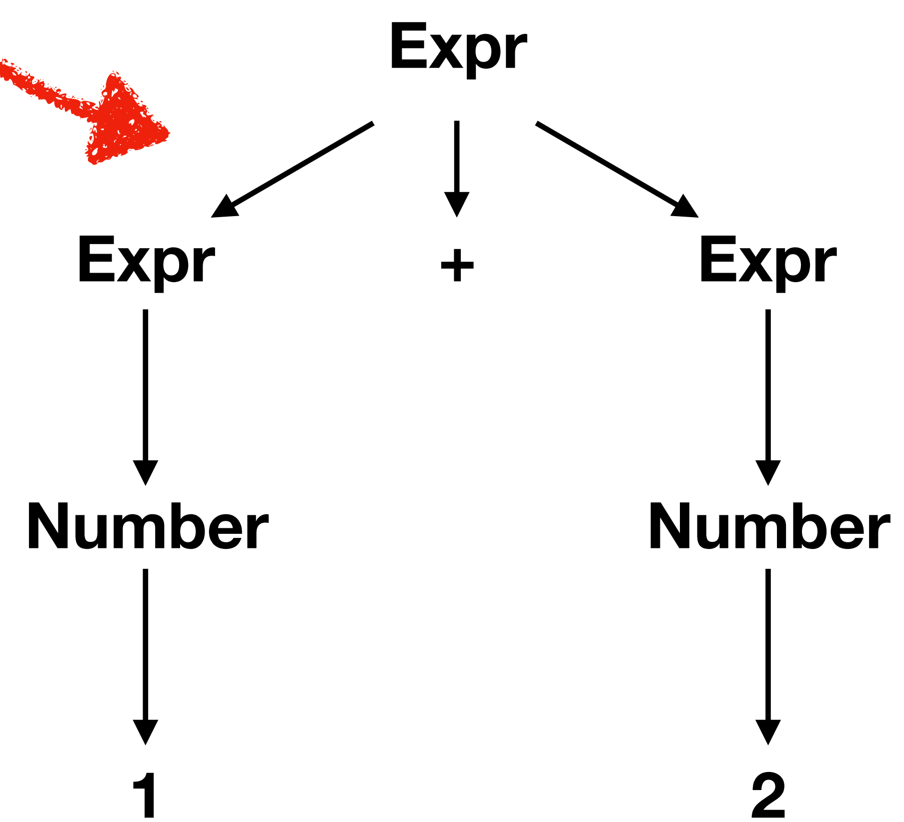
We will learn one fairly useful and easy-to-code one

(Recursive descent parsing, or LL(1) parsing)



```
1 + 2
```

```
(define (parse-input)
    ...)
```



Next week, we'll see how to **write** these parsers

Exercise: draw the parse trees for the following derivations

```
Expr
-> Expr + Expr
-> Expr + Expr * Expr
-> number + Expr * Expr
-> number + number * Expr
-> number + number * number
```

```
Expr
-> Expr * Expr
-> Expr + Expr * Expr
-> number + Expr * Expr
-> number + number * Expr
-> number + number * number
```

Here's an example of a grammar that is **not** ambiguous

```
Expr -> MExpr
Expr -> MExpr + MExpr
MExpr -> MExpr * MExpr
MExpr -> number
```

Generally, we’re going to want our grammar to be **unambiguous**

Question: Why are parse trees useful?

Answer: We can use them to define the meaning of programs

First, can represent parse trees in our PL:

```
(define my-tree
  '(+ 1 (* 2 3)))
```

This allows us to write **interpreters**

```
(define my-tree
  '(+ 1 (* 2 3)))

(define (evaluate-expr e)
  (match e
    [`(+ ,e1 ,e2) (+ (evaluate-expr e1) (evaluate-expr e2))]
    [`(* ,e1 ,e2) (* (evaluate-expr e2) (evaluate-expr e2))]
    [else e]))
```

Next lecture, we'll dig into grammars even more

Our goal is to write parsers, but to do so, we need more intuition about grammars