# Parsing

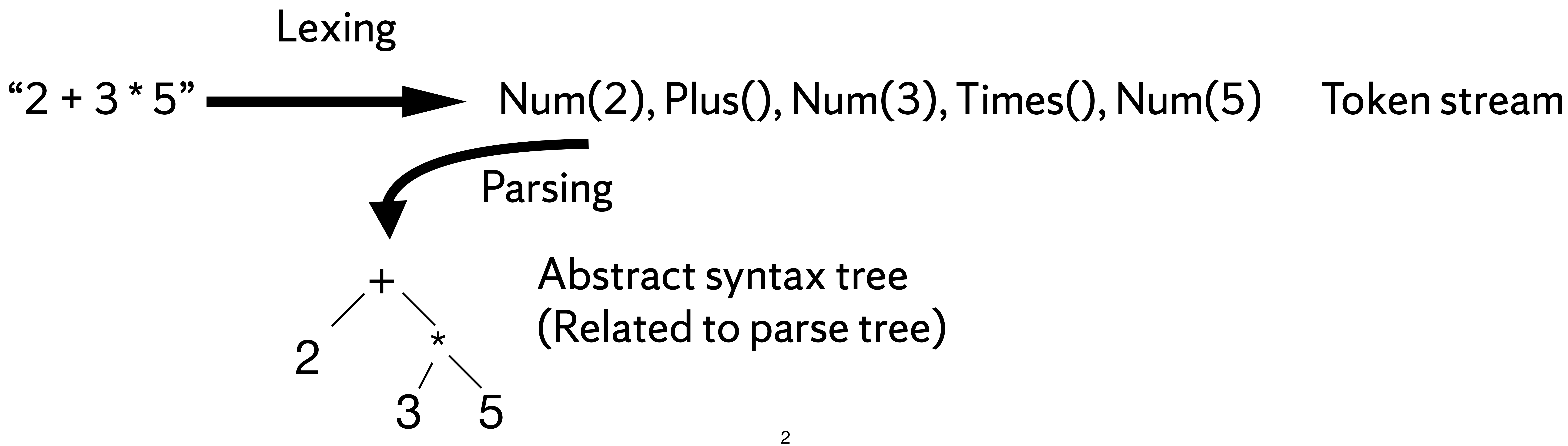
CIS531 — Fall 2025, Syracuse

Kristopher Micinski

Syracuse University
College of Engineering
& Computer Science

# This week: from raw bytes to an AST

- I find parsing rather boring: just use (read)!
  - But, some important concepts—and also, more time to practice Racket
- Two important concepts:
  - ***Tokenization*** (“lexing”), breaking the input stream (bytes) into logical “tokens”
    - Akin to what you do in LLMs, for example
  - ***Parsing***: fitting that stream of logical tokens to a hierarchical (tree-shaped) grammar



# The foundations of regular expressions

(Don't need to remember details)

# Introduction to grammars

(Important to get concepts)

# Lexical Analysis, Regular Expressions (regex)

- Generally, we specify individual tokens via *regular expressions*
- Regular expressions allow us to write string matchers from several patterns:
  - The empty string ε is a regular expression
  - Any literal can be matched, for example: “dog” is a regular expression
  - If e0 and e1 are regular expressions, so is e0 | e1 (matches either)
  - If e0 and e1 are regular expressions, so is e0e1 (concatenation)
  - If e is a regular expression, so is e*
    - This is the “Kleene star” and matches “0 or more” occurrences of e
- Practical implementations extend regexes to include other patterns
  - Also, common implementations *fundamentally extend regex power*
    - E.g., superlinear regular expressions; crashed the internet several times!!

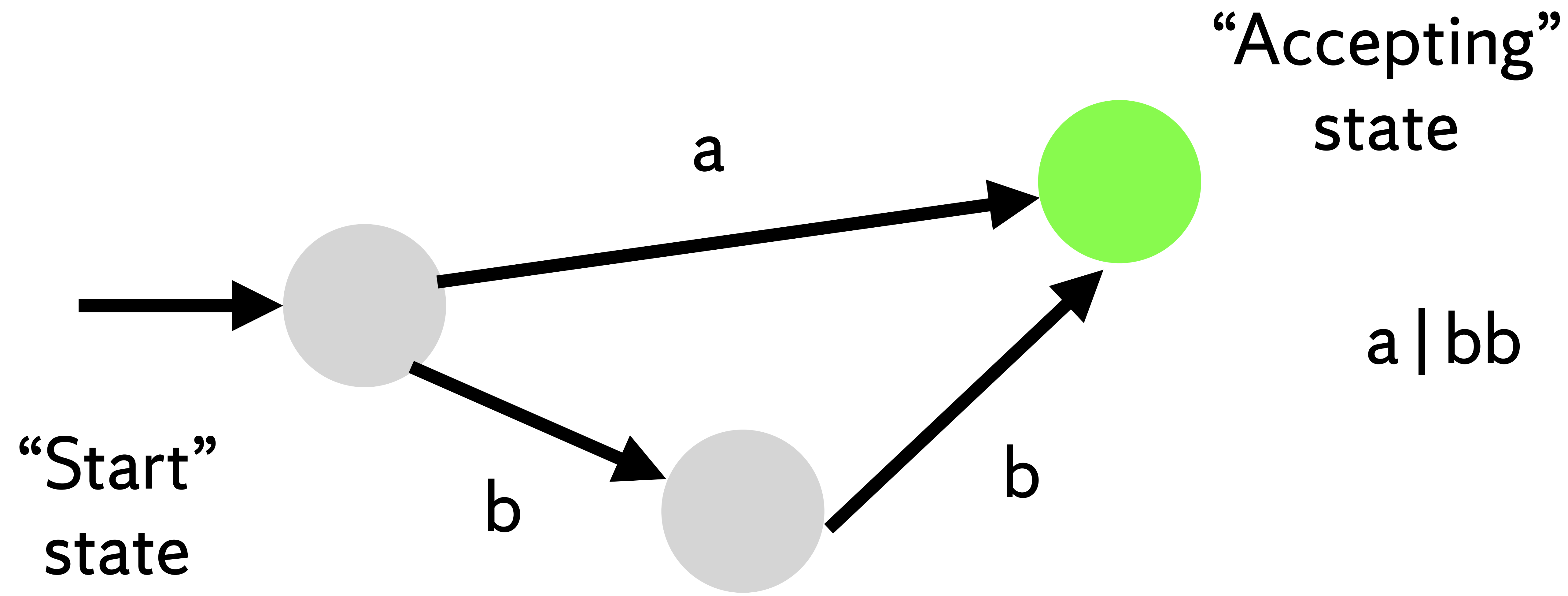
# RE example 1

- Which of the following strings is matched by the regex “ab*cd*”
  - ac
  - abbbccdddd
  - abbbacddd
  - abcd
  - acdddddda

- Which of the following strings is matched by the regex “he(l||ll)(o|p)
  - hello
  - hll
  - help
  - hellp

# How are REs implemented? (Abbreviated version...)

- Every RE can be systematically compiled to a nondeterministic finite automaton




- Every NFA can be further compiled to a DFA, implemented via a lookup table

# Finite Automata Example

- Write an NFA for the regular expression a(bc)*d

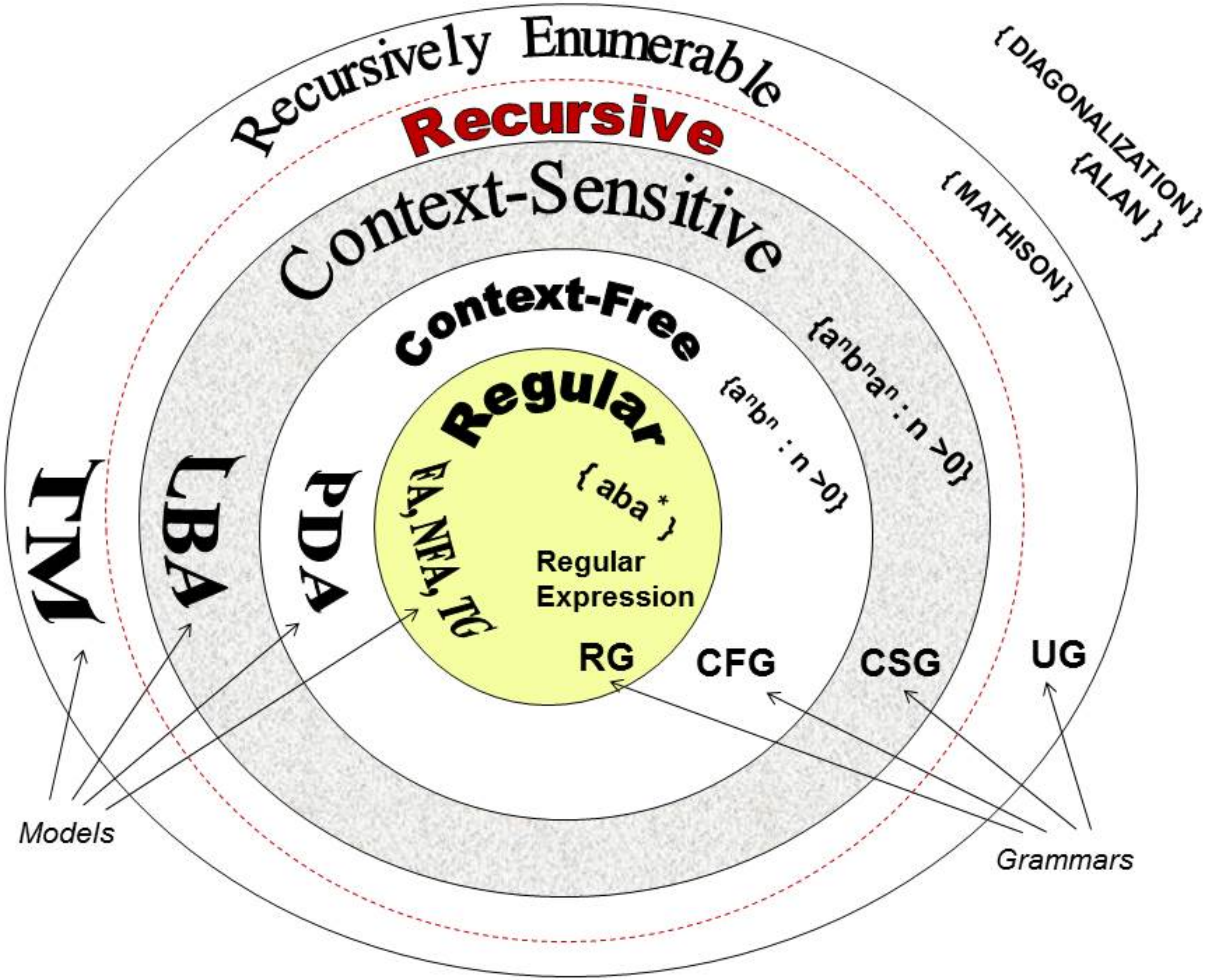
```
(define-lex-abbrev WS     (:+ (char-set " \t\r\n")))
(define-lex-abbrev DIGITS (:+ numeric))

(define expr-lexer
  (lexer
    [WS          (expr-lexer input-port)]
    ["+"         'PLUS]
    ["-"         'MINUS]
    ["*"         'TIMES]
    ["("         'LPAREN]
    [")"         'RPAREN]
    ["read"      'READ]
    ["print"     'PRINT]
    [DIGITS      `(INT ,(string->number lexeme))]
    [(eof)       'EOF]))

(define (tokenize-port in-port)
  (let loop ([acc '()])
    (define t (expr-lexer in-port))
    (if (eq? t 'EOF)
        (reverse (cons t acc))
        (loop (cons t acc)))))

;; Tokenize the string, turning it into a list of tokens.
(define (tokenize-string str) (tokenize-port (open-input-string str)))

;; (pretty-print (tokenize-port (open-input-string "3 + 3 * 5")))
;; (pretty-print (tokenize-string "3 + 3 * 5"))
```

# Lexing vs. Parsing

- Lexing is relatively "easy:"
  - Specify tokens via regular expressions, REs readily translate into finite automata
  - Well-established results tell us of the equivalence of NFAs, DFAs, subset construction, etc.
  - In other words, REs are ***fast*** (in principle, at least—when avoiding nonlinear features)
- By contrast, parsing is harder:
  - Can't specify most language constructs via REs—e.g., balanced parentheses
  - These require *context-free grammars* (CFGs) which are strictly more powerful than RE
  - Sometimes easy to implement (e.g., LL(k) grammars, predictive parsing), but in general may require using a parser generator (e.g., bottom-up parsing)
  - New innovations, even to this day—but only basic knowledge required for day-to-day usefulness, I argue
    - Often using some well-known format anyway (JSON, S-expressions, etc.)

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?

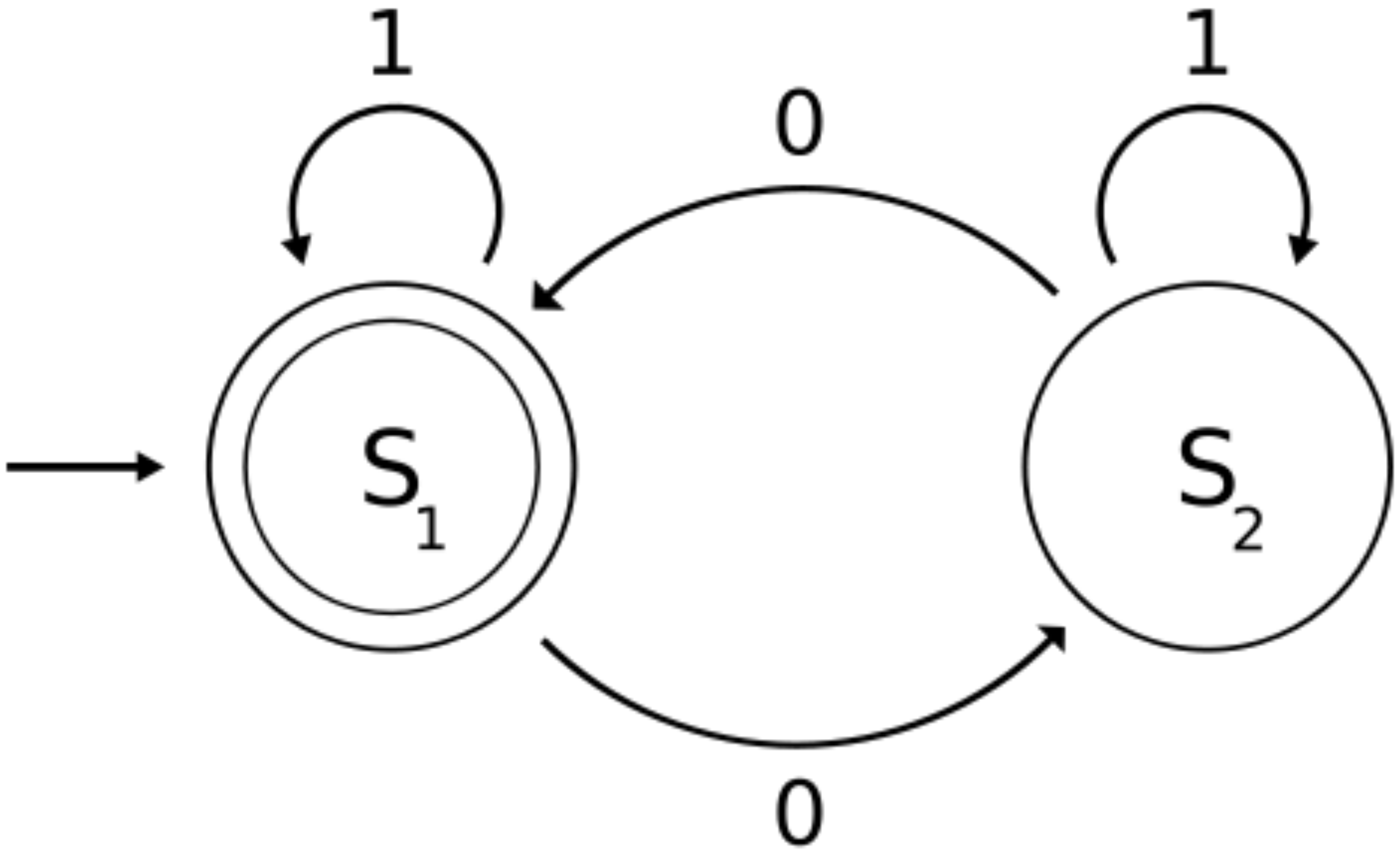
“The set of strings P such that P…?”

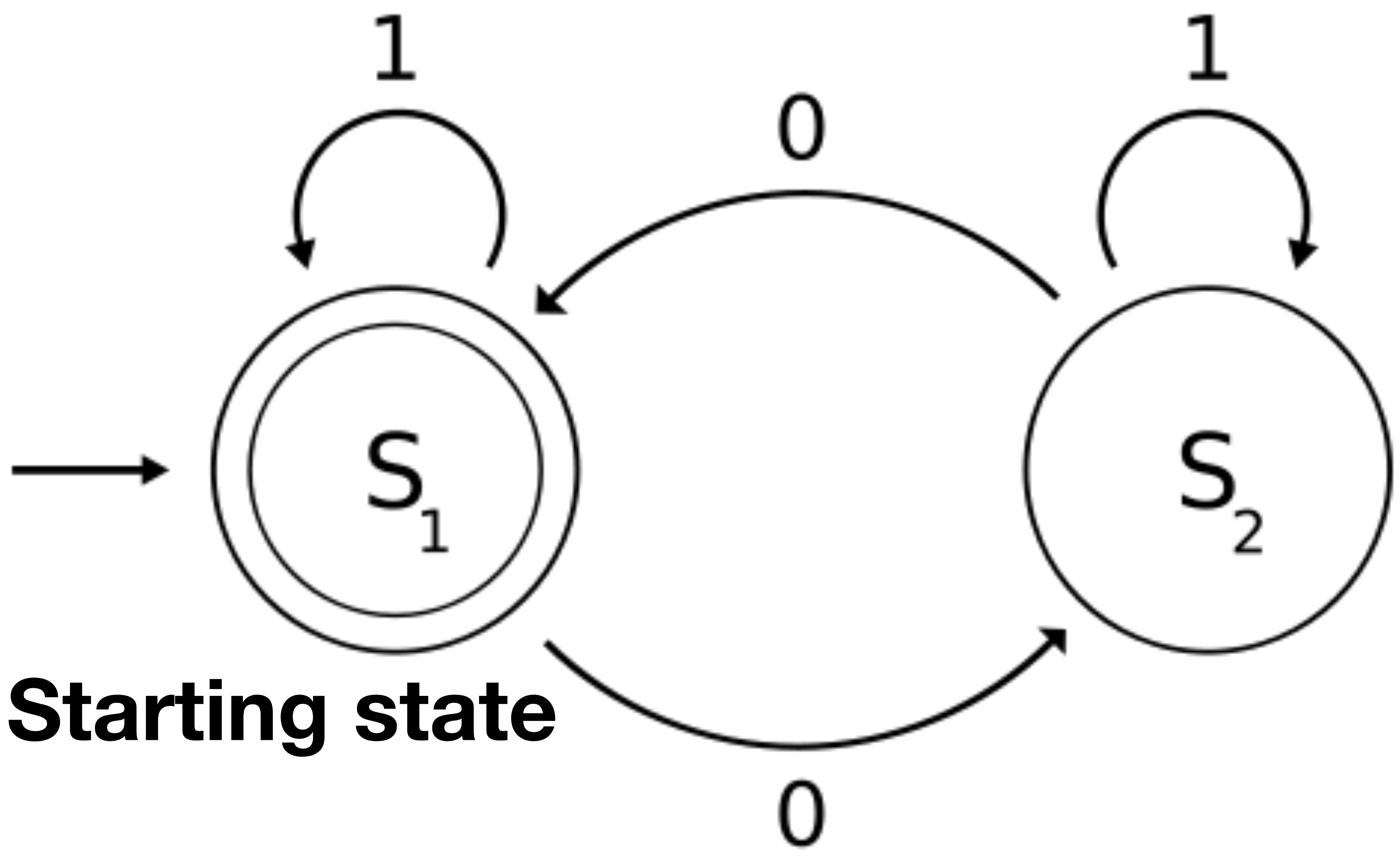
(**Answer**: is a program that halts)

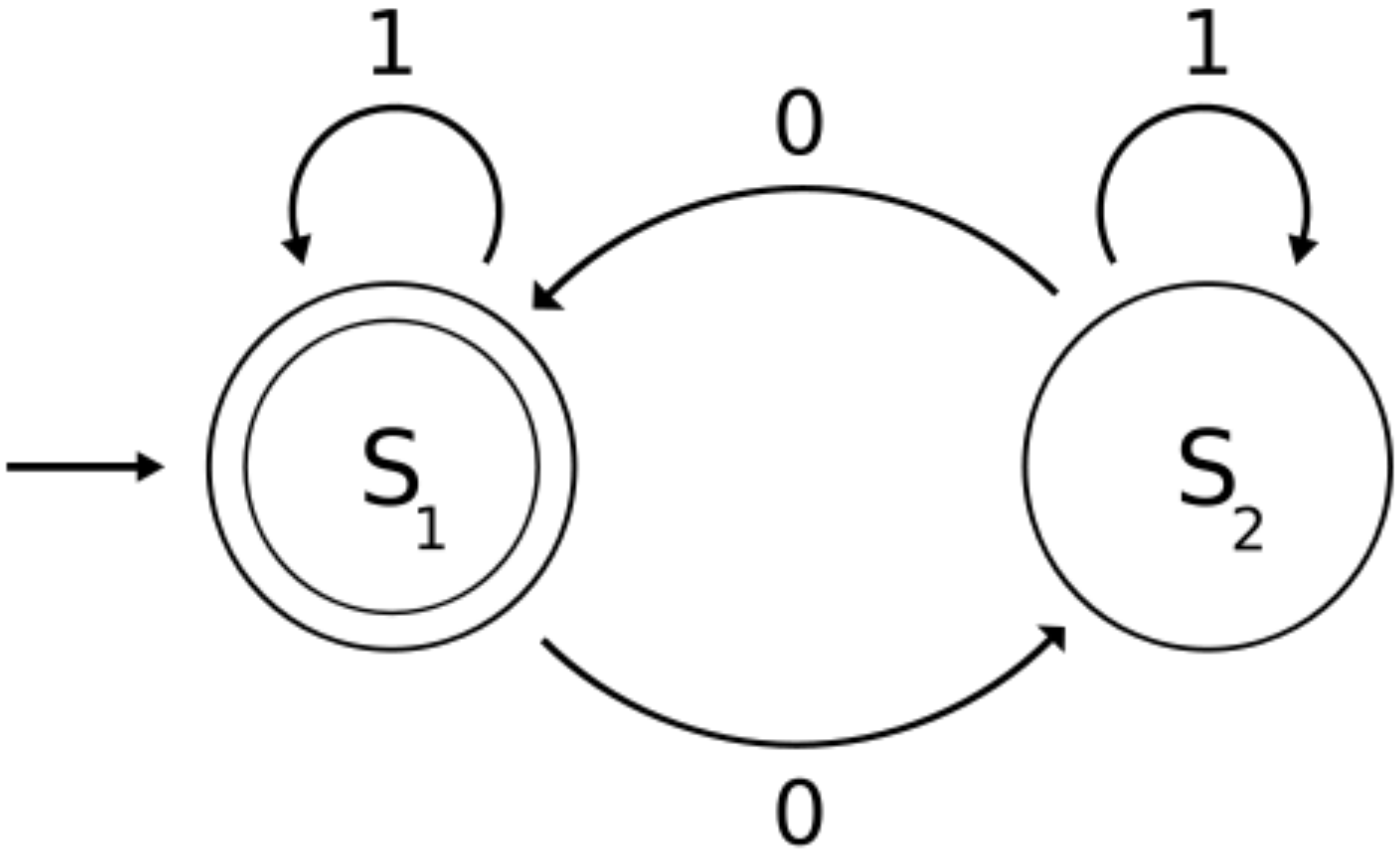
Regular expressions can be **implemented** using **finite state machines**

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

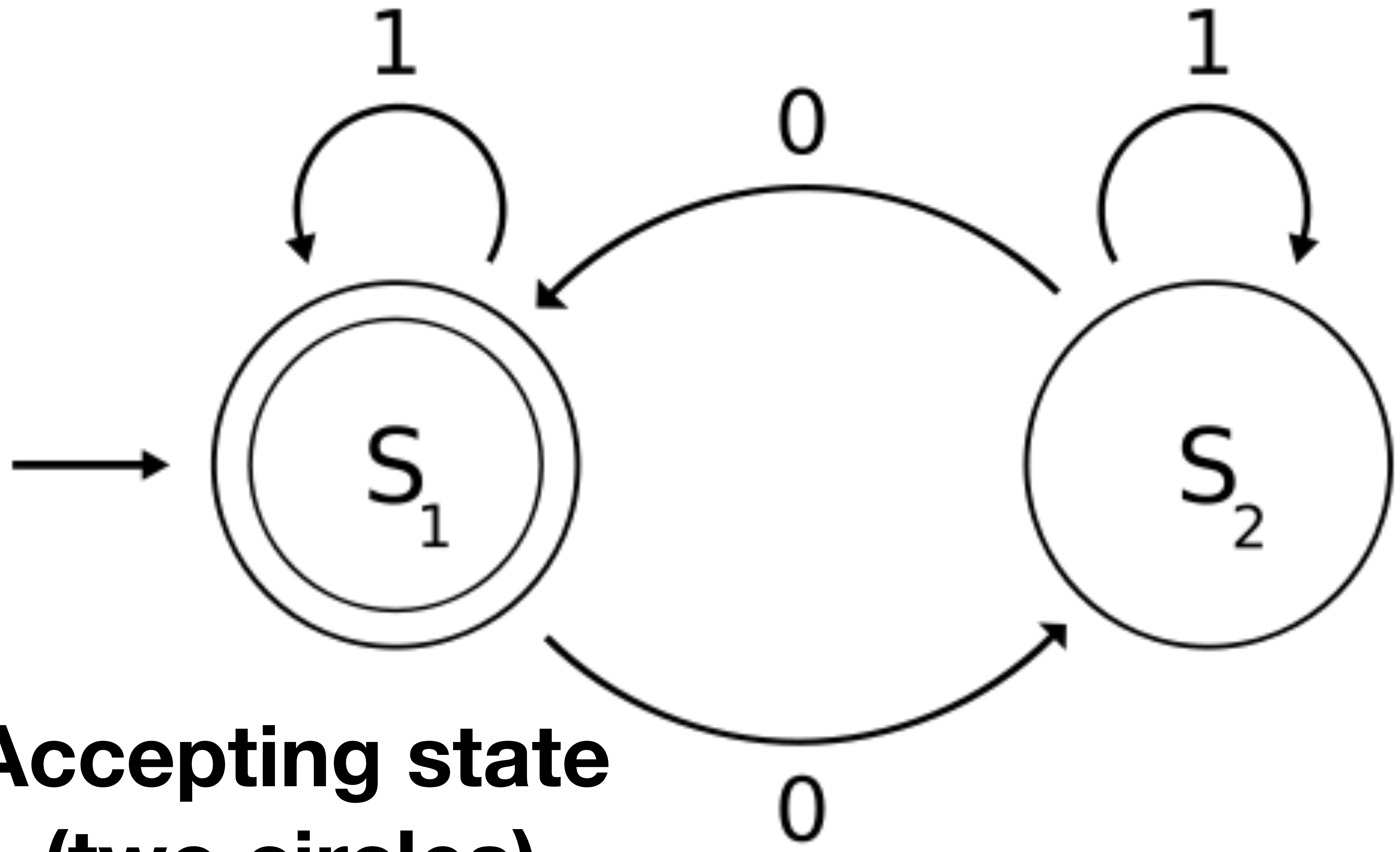
All regexes can “compile” (turn to, in systematic way) FSM


1
0
1
S1
S2
0


1
1
0
S1
S2
Starting state
0

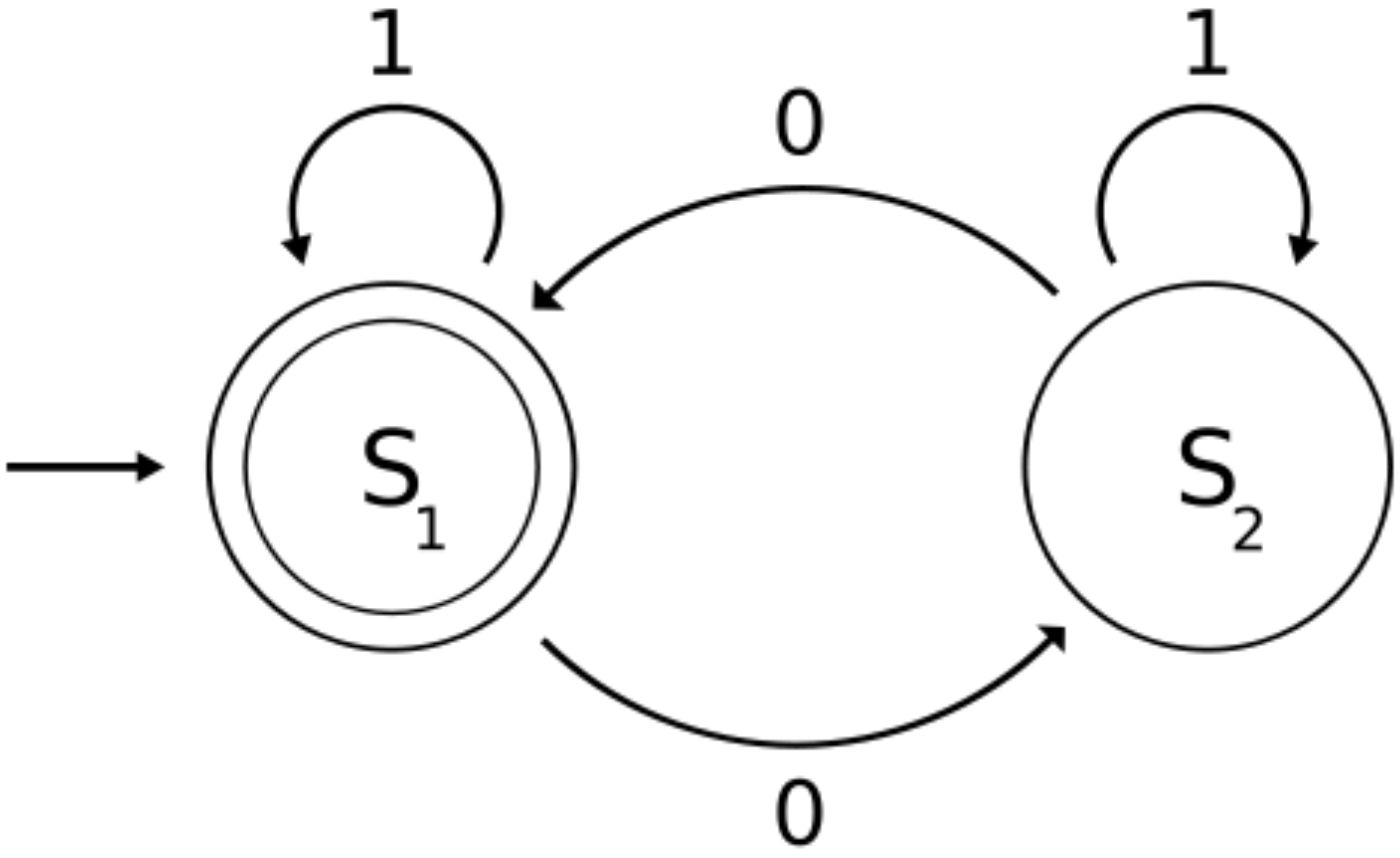



**Transition on input**

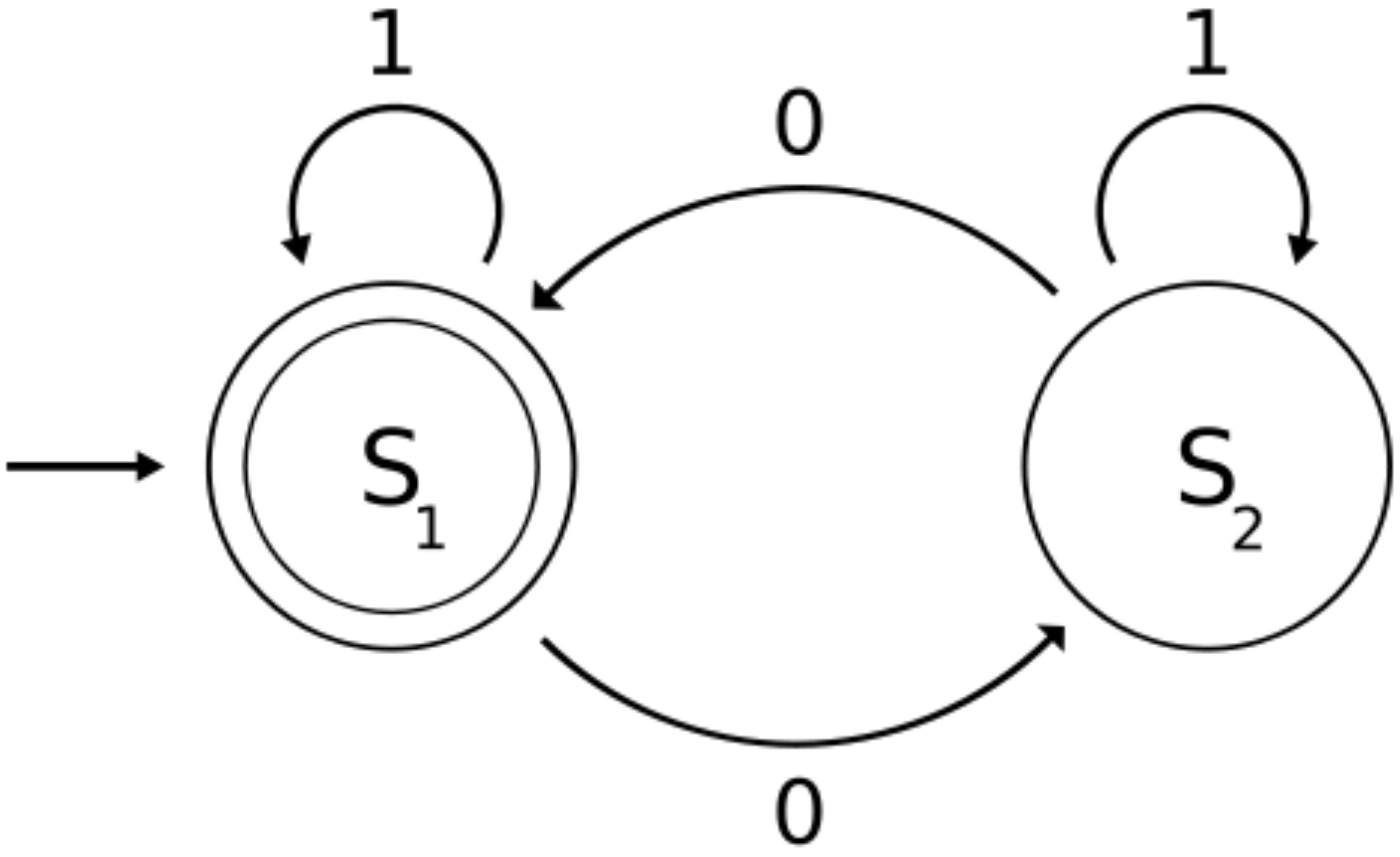



**Accepting state (two circles)**

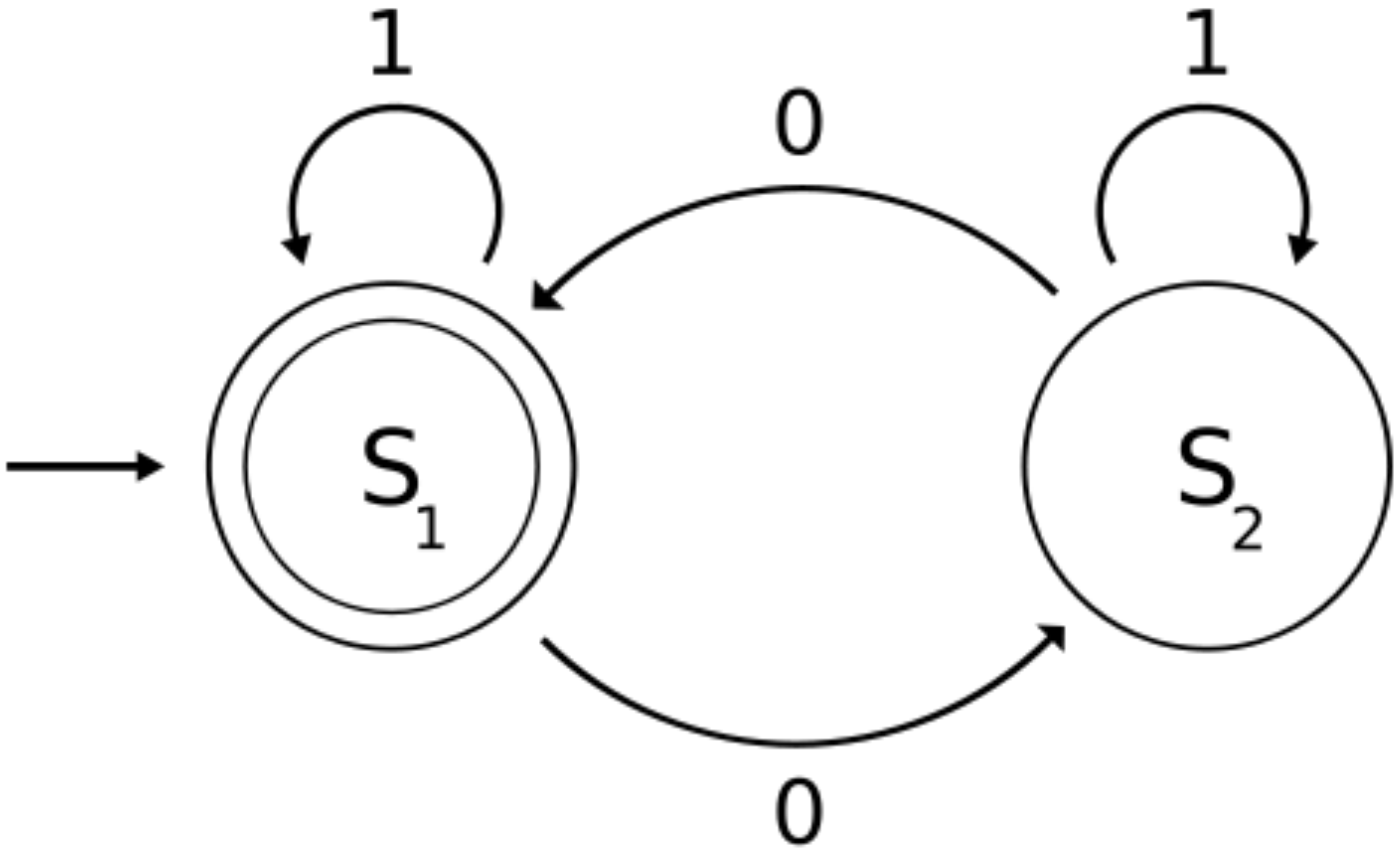
**011** S1



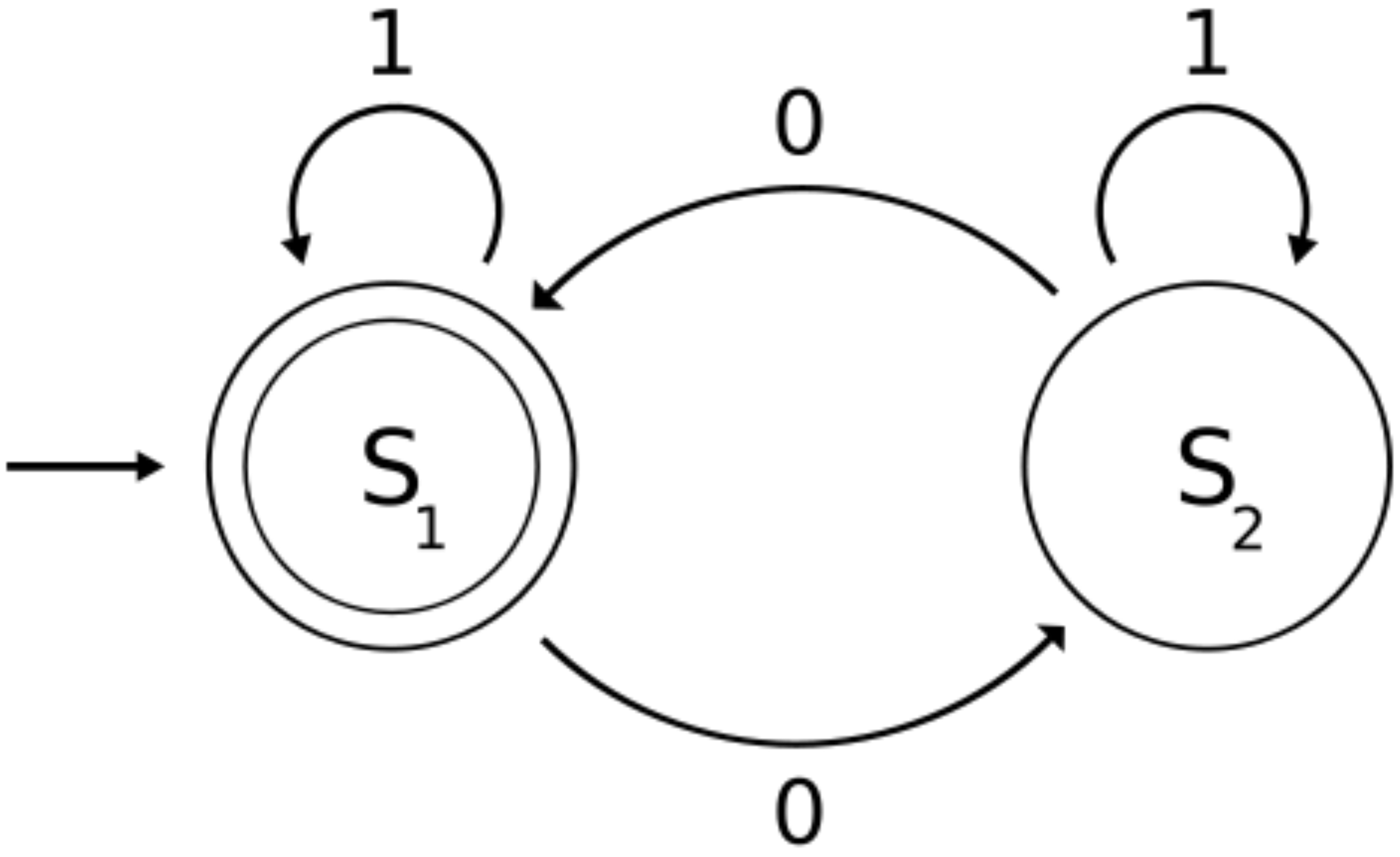
**0̲11** S2



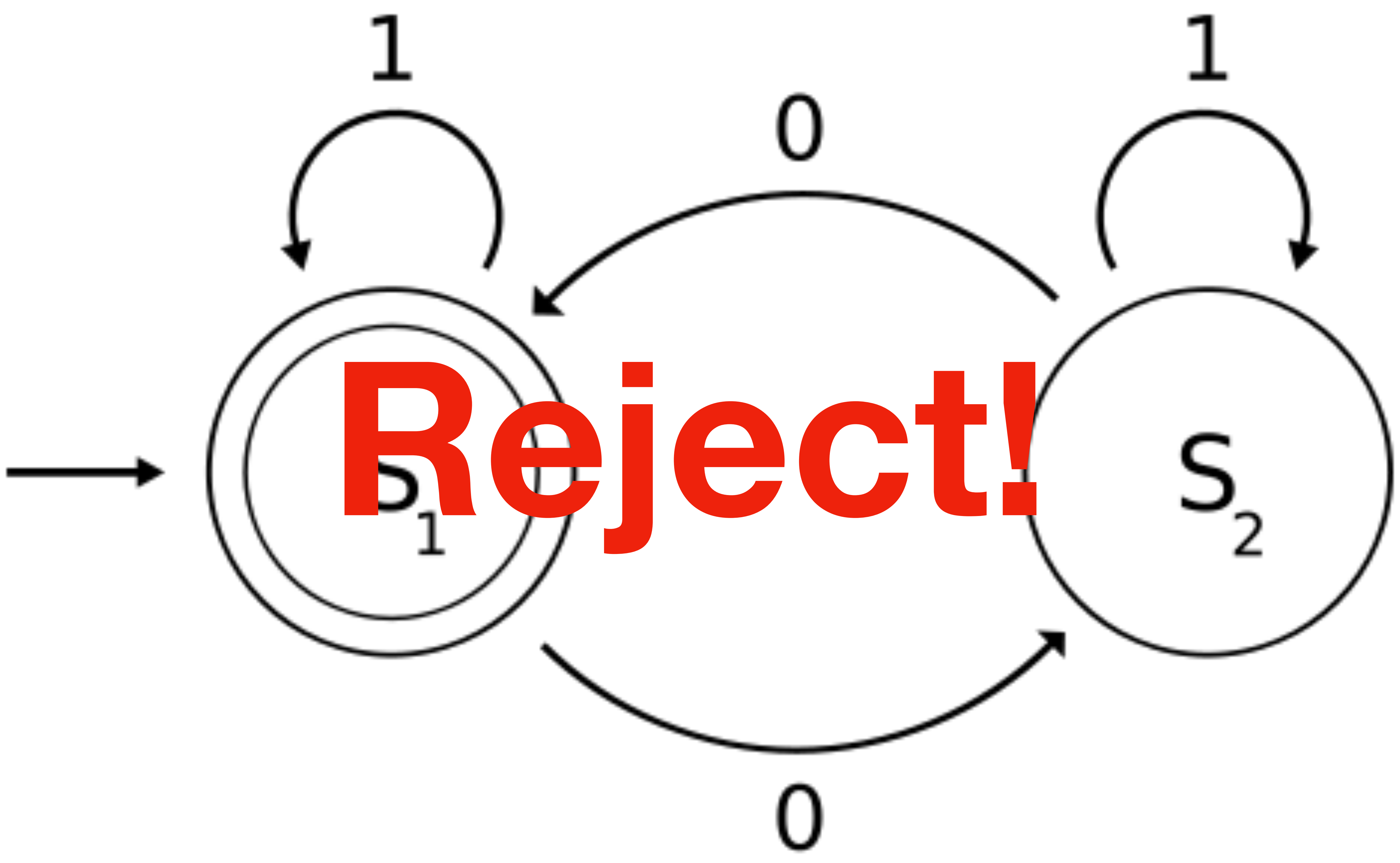
**011** S2 **Stay!**



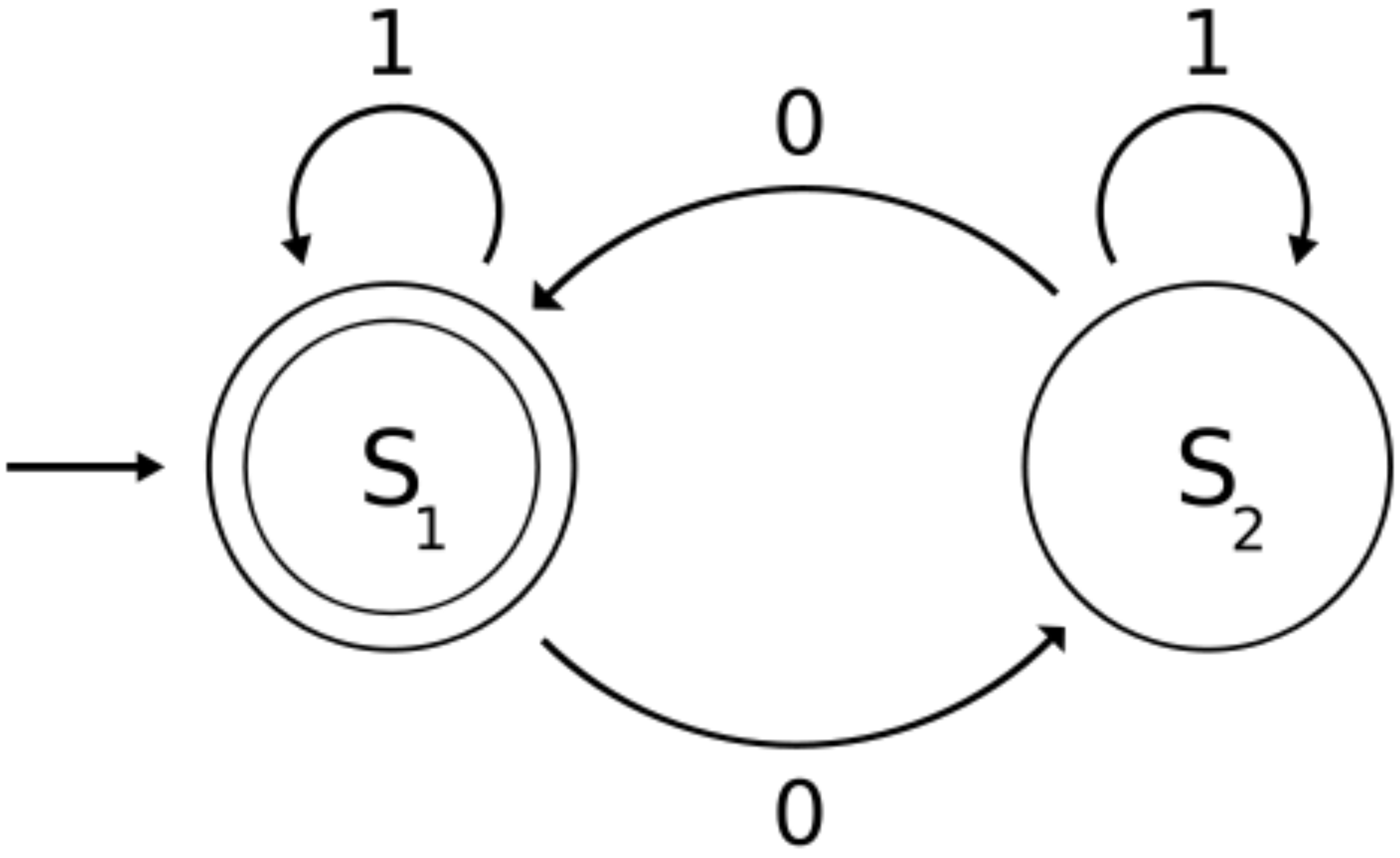
**<u>011</u>** S2



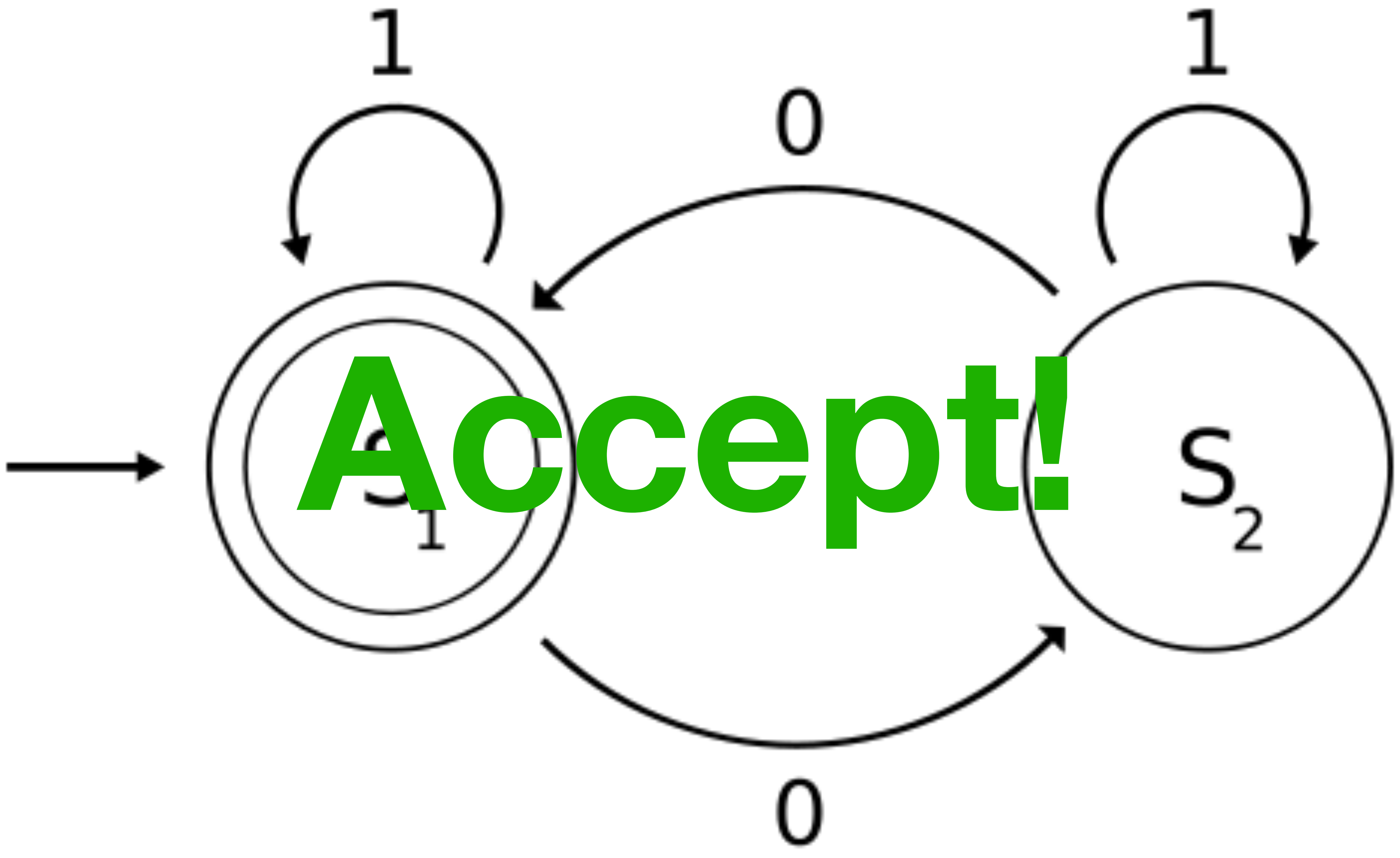
**<u>011</u>** S2



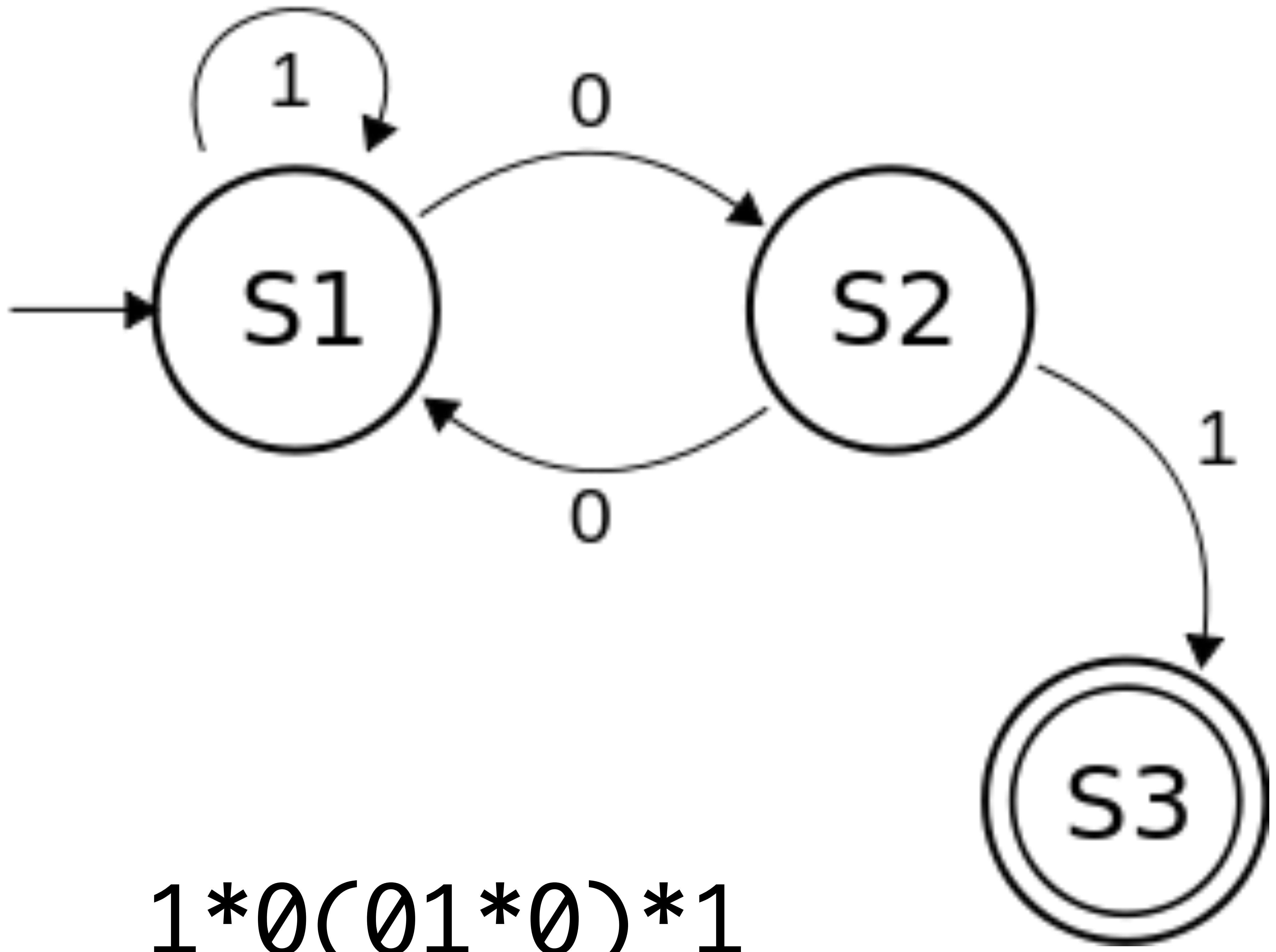
**<u>0110</u>** S1



**0110** S1



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




1*0(01*0)*1

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

# Beyond lexing: parsing

- Can we use regular expressions to match a whole language..?
- No! Interesting languages can not be written as **any** regex
- Examples include: balancing parentheses, if/then/else
  - Anything where the program would need to “count” (to an arbitrary degree)
  - Counting is beyond the power of finite automata—also need a *stack*
    - Pushdown automata, context-free languages, etc…

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

```
{}

{{}}

{{{}}}

{{{{}}}}
```

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

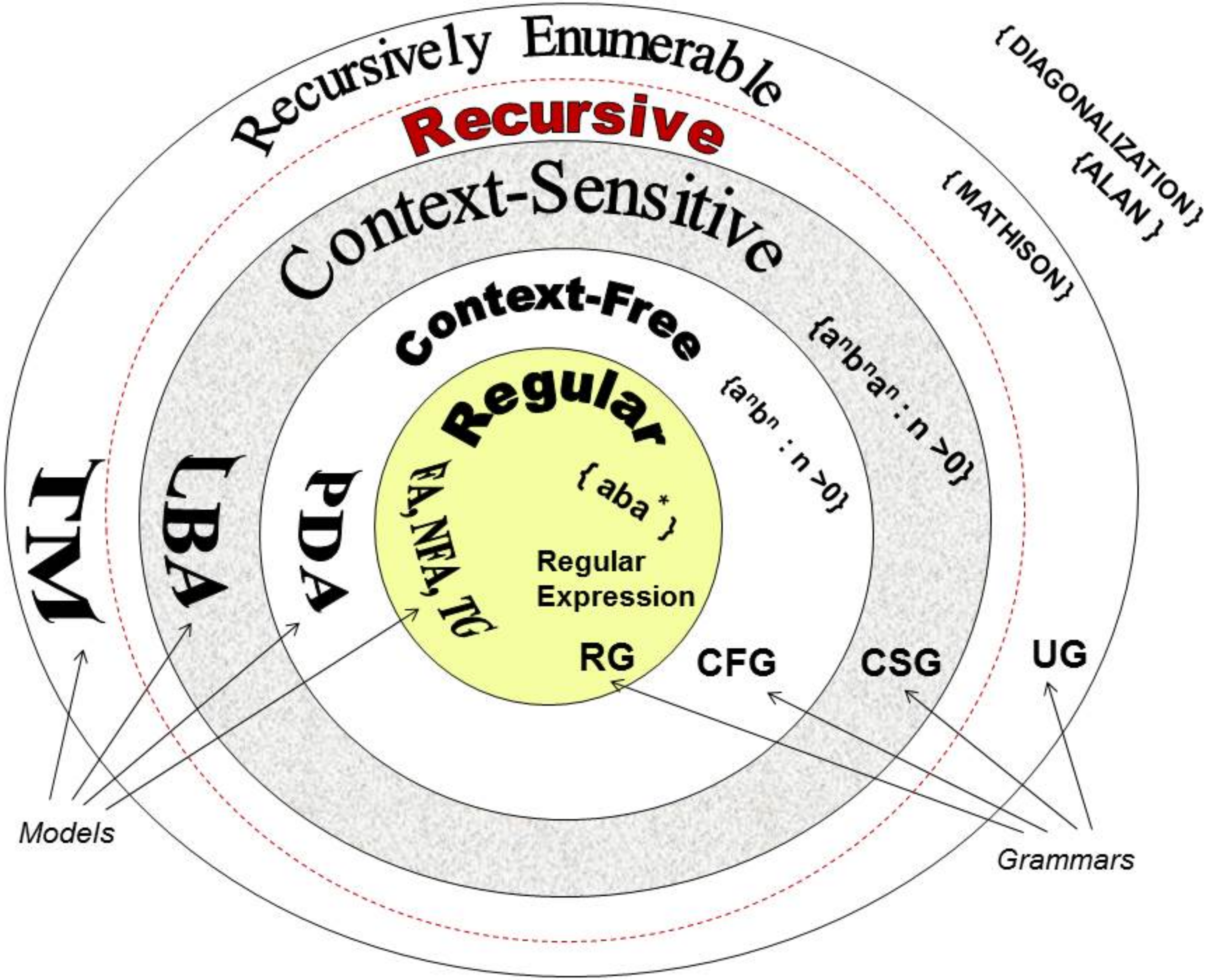
{*}* doesn’t work

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 }
```



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

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**

# Context-Free Grammars

- CFGs (context-free grammars) generalize REs
    - Any RE can be written as a CFG
- Below is an example of a grammar for expressions…

```
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

# The “meaning” of a CFG. Definition: derivation

- To determine if a grammar matches an expression, **you play a game**
- Start by writing down the start symbol
- Continue by expanding a nonterminal according to one of the productions
- This trace (sequence of partial steps) is called a ***derivation***

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
```

This is a “complete” derivation because it ends in a terminal string (only terminals left)

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
???
```

This is an "incomplete" derivation because it gets stuck—but each step follows the rules

```
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

```
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 …
```

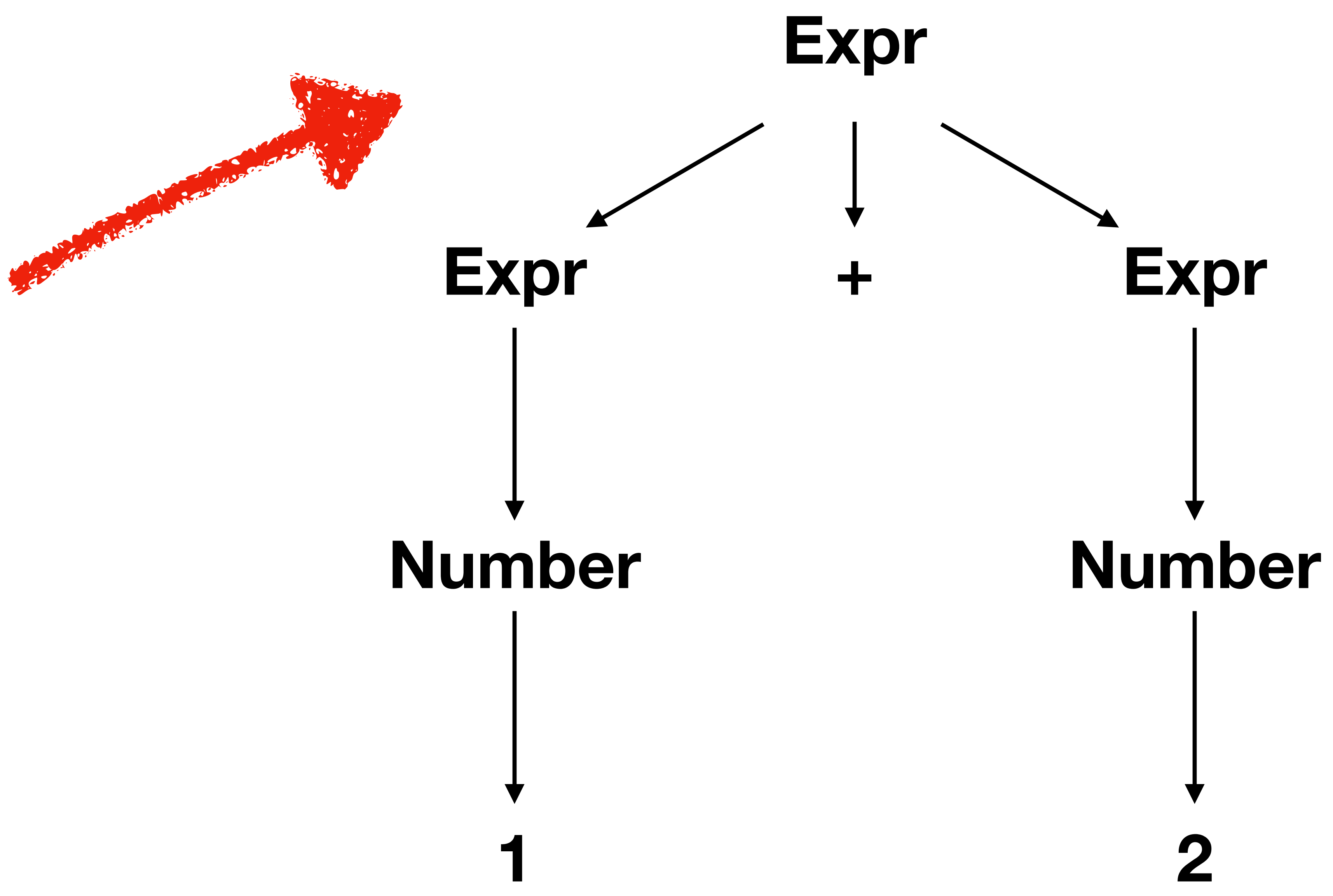
Does the else belong to the first if? Or the second?

(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

**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
```

**Question**: Why are parse trees useful?

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

# Parsing *Algorithms*

- Although we can write down a grammar, and I can define to you what a derivation is, this doesn't immediately yield a parsing **algorithm**
  - "Here's a grammar, now go *find* a derivation"
- The goal of a parsing algorithm is to take your grammar and realize it as a program that says either:
  - (a) YES, the string matches, and here's a parse tree
  - (b) NO, the string doesn't match, (maybe) and here's where I got stuck
- Some grammars are **easy** to parse, some will be **harder**

# General idea: why not try *everything possible!*?

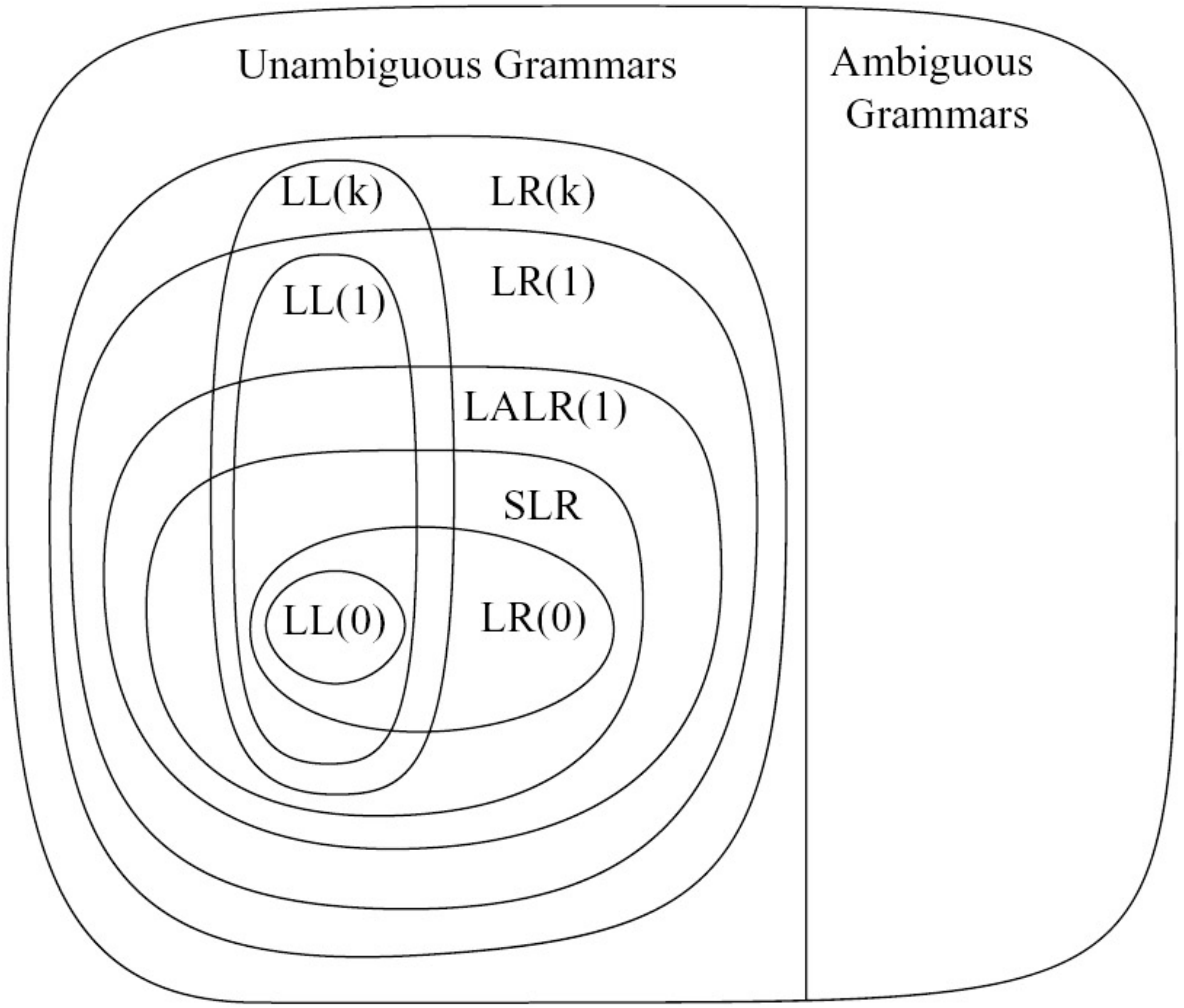
- In a certain way, parsing is not too bad:
  - "Just simulate *all possible derivations in parallel*"
    - The famous **Early** parsing algorithm does this—it is **very** general
      - But what is the downside? Answer: very high computational complexity!
    - Other variants:
      - CYK, GLR parsing, LL(*) with infinite backtracking

- For most grammars, we can exploit some of the structure to do better…

There are a lot of different parsing algorithms, I will focus on the simpler case…

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

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



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

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

# Two kinds of derivations

**Leftmost derivation**: The leftmost nonterminal is expanded first at each step

**Rightmost derivation**: The rightmost nonterminal is expanded first at each step

```
G -> GG
G -> a
```

Draw the **leftmost derivation** for…

*aaa*

Draw the **rightmost derivation** for…

*aaa*

```
G -> G + G
G -> G / G
G -> number
```

Draw a leftmost derivation for…

```
1 / 2 / 3
```

Now draw *another* leftmost derivation

Draw the parse trees for each derivation

What does each parse tree mean?

A grammar is **ambiguous** if there is a string with **more than one** leftmost derivation

(Equiv: has more than one parse tree)

Parsing algorithms require that our grammar be ***unambiguous***

(If not, the parser has to return a set of derivations)

```
G -> G + G
G -> G / G
G -> number
```

**There’s another problem with this grammar (OOO)**

We need to tackle **ambiguity**

Idea: introduce extra nonterminals that force you to get left-associativity

(Also force OOP)

```
Add -> Add + Mul | Mul
Mul -> Mul / Term | Term
Term -> number
```

Write derivation for 5 / 3 / 1

Draw the parse tree for 5 / 3 / 1

```
Add -> Add + Mul | Mul
Mul -> Mul / Term | Term
Term -> number
```

This grammar is **left recursive**

```
Add -> Add + Mul | Mul
Mul -> Mul / Term | Term
Term -> number
```

A grammar is left-recursive if any nonterminal A has a production of the form A -> A…

```
Add -> Add + Mul | Mul
Mul -> Mul / Term | Term
Term -> number
```

This will turn out to be bad for one class of parsing algorithms

Let's say I want to parse the following grammar

```
S -> aSa | bb
```

# Recursive Descent and LL(k) parsing

- In a recursive descent parser, often called a "predictive parser," I translate my grammar into a **set of recursive functions** which use *lookahead* to *predict* which branch of the derivation needs to be taken.
- Each nonterminal E is translated into a function, `parse_E`

```
E -> b | cAc

A -> aAa | d
```

```
def parse_E():
  if (next_tok() == “b”):
    consume(“b”)
    return
  elif (next_tok() == “c”):
    consume(“c”)
    parse_A()
    consume(“c”)
    return
```

```
def parse_A():
  if (next_tok() == “a”):
    consume(“a”)
    parse_A()
    consume(“a”)
    return
  elif (next_tok == “d”):
    consume(“d”)
    return
```

# First, a few questions

S -> aSa | bb

Is this grammar ambiguous?

If I were matching the string **bb,** what would my derivation look like?

If I were matching the string **abba,** what would my derivation look like?

# First, a few questions

S -> aSa | bb

**Key idea: if I look at the next input, at most one of these productions can “fire”**

If I see an **a** I **know** that I **must** use the first production

If I see a b, I know I must be in second production

Slight transformation..

```
S -> A | B
A -> aAa
B -> bb
```

Slight transformation..

```
S -> A | B
A -> aAa
B -> bb
```

Now, I write out **one function** to parse **each** nonterminal

# FIRST(A)

FIRST(A) is the **set** of terminals that could occur **first** when I recognize A

Note: ε **cannot** be a member of FIRST because it is not a character

# NULLABLE

Is the set productions which could generate ε

# FOLLOW(A)

FOLLOW(A) is the set of terminals that appear immediately to the right of A in some form

```
S -> A | B
A -> aAa
B -> bb
```

**What is FIRST for each nonterminal**

**What is NULLABLE for the grammar**

**What is FOLLOW for each nonterminal**

More practice…

```
E  → TE'
E' → +TE'
E' → ε
T  → FT'
T' → *FT'
T' → ε
F  → (E)
F  → id
```

**What is FIRST for each nonterminal**

**What is NULLABLE for the grammar**

**What is FOLLOW for each nonterminal**

Let's say I want to parse S

```
A -> aAa | B
B -> bb
```

I **look at the next token**, and I have *two possible choices*

If I see an **a**, I must parse an A

If I see a **b**, I must parse a B

We use the **FIRST** set to help us design our recursive-descent parser!

Livecoding this parser in class

The recursive-descent parsers we will cover are generally called **predictive** parsers, because they use **lookahead** to predict which production to handle next

# LL(1)

A grammar is LL(1) if we only have to look at the **next** token to decide which production will match!

I.e., if S -> A | B, FIRST(A) $\cap$ FIRST(B) must be empty

L eft to right

L eft derivation

1 token of lookahead

Recursive-descent is called **top-down** parsing because you build a parse tree from the root down to the leaves

There are also **bottom-up** parsers,
which produce the rightmost derivation

Won't talk about them, in general they're impossibly-hard to write / understand, easier to use

What about this grammar?

```
E -> E - T | T
T -> number
```

This grammar is **left recursive**

```
E -> E - T | T
T -> number
```

What happens if we try to write recursive-descent parser?

Infinite loop!

We can **remove left recursion**

```
E -> E - T | T
T -> number
```

**Factor!**

```
E  -> T E'
E' -> - T E'
E' -> ε
```

# In general, if we have

```
A -> Aα | bB
```

## Rewrite to…

```
A -> bB A'
A' -> α A' | ε
```

Generalizes even further

https://en.wikipedia.org/wiki/LL_parser#Left_Factoring

## But this still doesn't give us what we want!!!

```
E  -> T E'
E' -> - T E'
E' -> ε
```

```
E -> T E'
  -> T - T E'
  -> T - T - T E'
  -> T - T - T
```

# So how do we get left associativity?

**Answer: Basically, stupid hack in implementation**

```
Sub -> num Sub’
Sub’ -> + num Sub’ | epsilon
```

Is basically…

```
Sub -> num Sub’ (+ num)*
```

Intuition: treat this as while loop, then when building parse tree, put in left-associative order

```
Sub -> num Sub’ (+ num)*
```

```
Sub -> num Sub’
Sub’ -> + num Sub’ | epsilon
```

# LR (shift/reduce) parsing

- We did not talk much about the other large class of parsing algorithms, LR parsers
- LR(k) parsers construct the ***rightmost*** derivation, working left-to-right
    - Nice advantage—no issue with left recursion in grammars!
    - (Handle associativity properly, no factoring/tricks)
- **Key idea:** maintain a stack of symbols (terminals / nonterminals)
    - At every (next) input, you can either shift onto the stack, or reduce the stack by applying a transformation via two tables:
    - *Action table*: shift, reduce, accept, error
    - *Goto* table: jump post-reduction
- 👍 — works for most languages you'd want to write, fast to implement
- 👎 — requires a parser generator (tables are too tedious to do by hand for any nontrivial language), shift/reduce, reduce/reduce conflicts are hard to debug!

# Parsing: Fin

- My goal was to give you the basics of grammars, along with their key properties and transformations. Can you define: grammar, LL(k), LR, recursive descent?
- What to know / practice: could you write a simple recursive-descent parser?
- One exam problem (making clear now): given some relatively simple grammar, can you write a recursive descent parser?
  - You can use any language—if you want to use pseudocode, fine, as long as I can get the idea