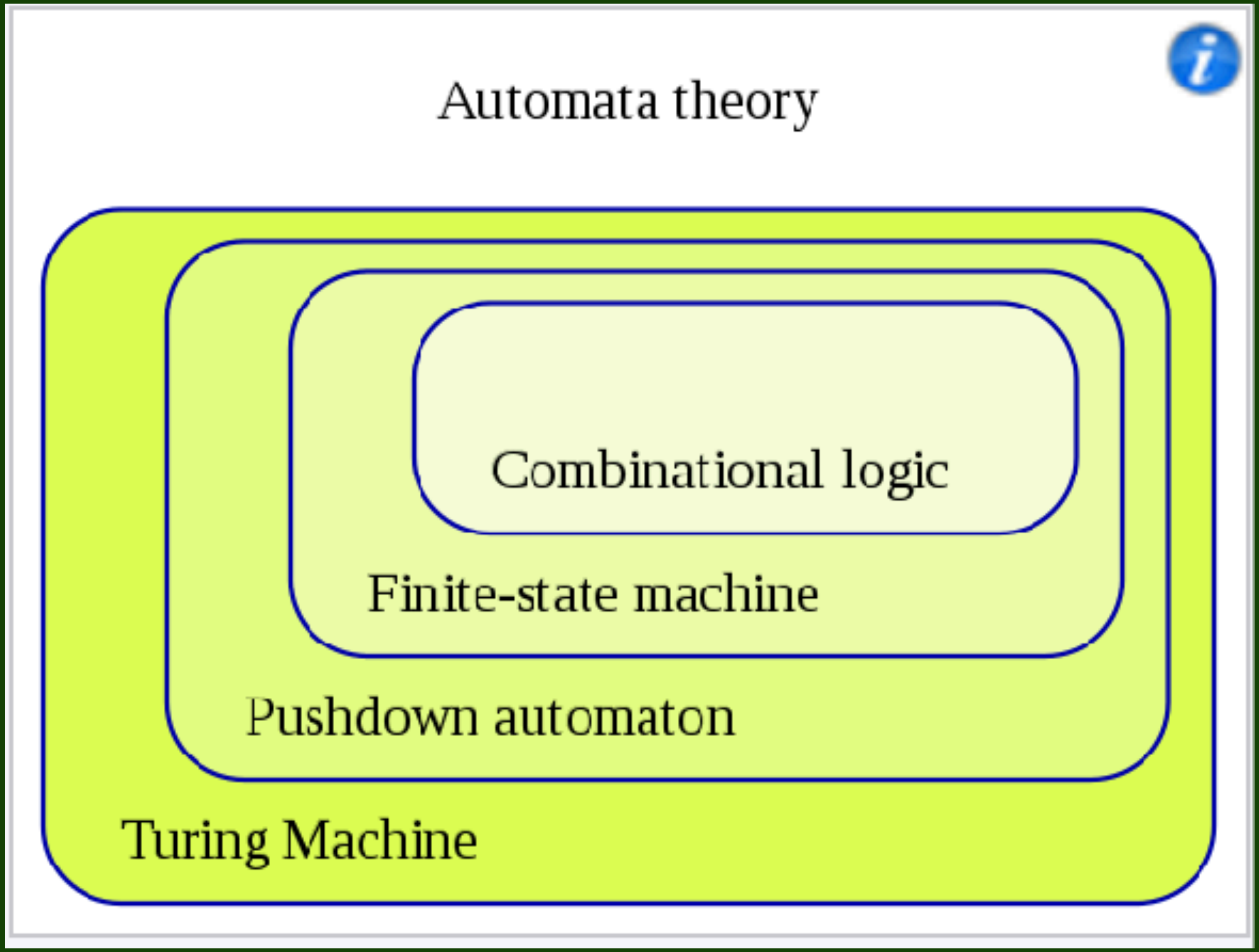
# Finite Automata, Regular Expressions

# What are automata?

**Automata theory** is the study of abstract machines and automata, as well as the computational problems that can be solved using them. It is a theory in theoretical computer science and discrete mathematics (a subject of study in both mathematics and computer science). The word *automata* (the plural of *automaton*) comes from the Greek word αὐτόματα, which means "self-acting".

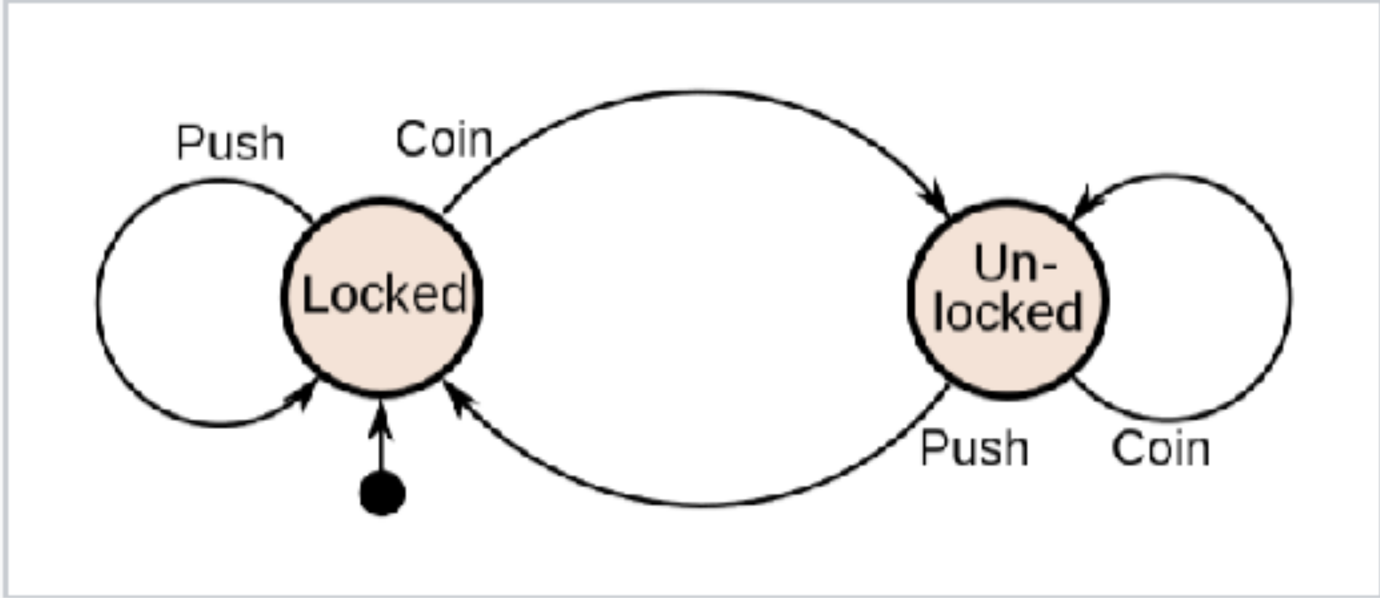
# Automata




Automata are further divided into deterministic and non-deterministic

# Finite state machine

- A simple two-state FSM: a turnstile (like the gates going on to the “T”)—substitute “Charlie Card” for coin:




State diagram for a turnstile

# FSMs, etc.

- Finite state machines have limited “memory”—only the various states. In order to read realistic regular expressions, we need “push-down” automata.
- Push-down automata essentially give us a stack where we can manipulate the stack in addition to the various transitions (but we can’t look inside the stack).
- Turing machine gives us essentially unlimited memory to store results/states.

# Regular Expressions

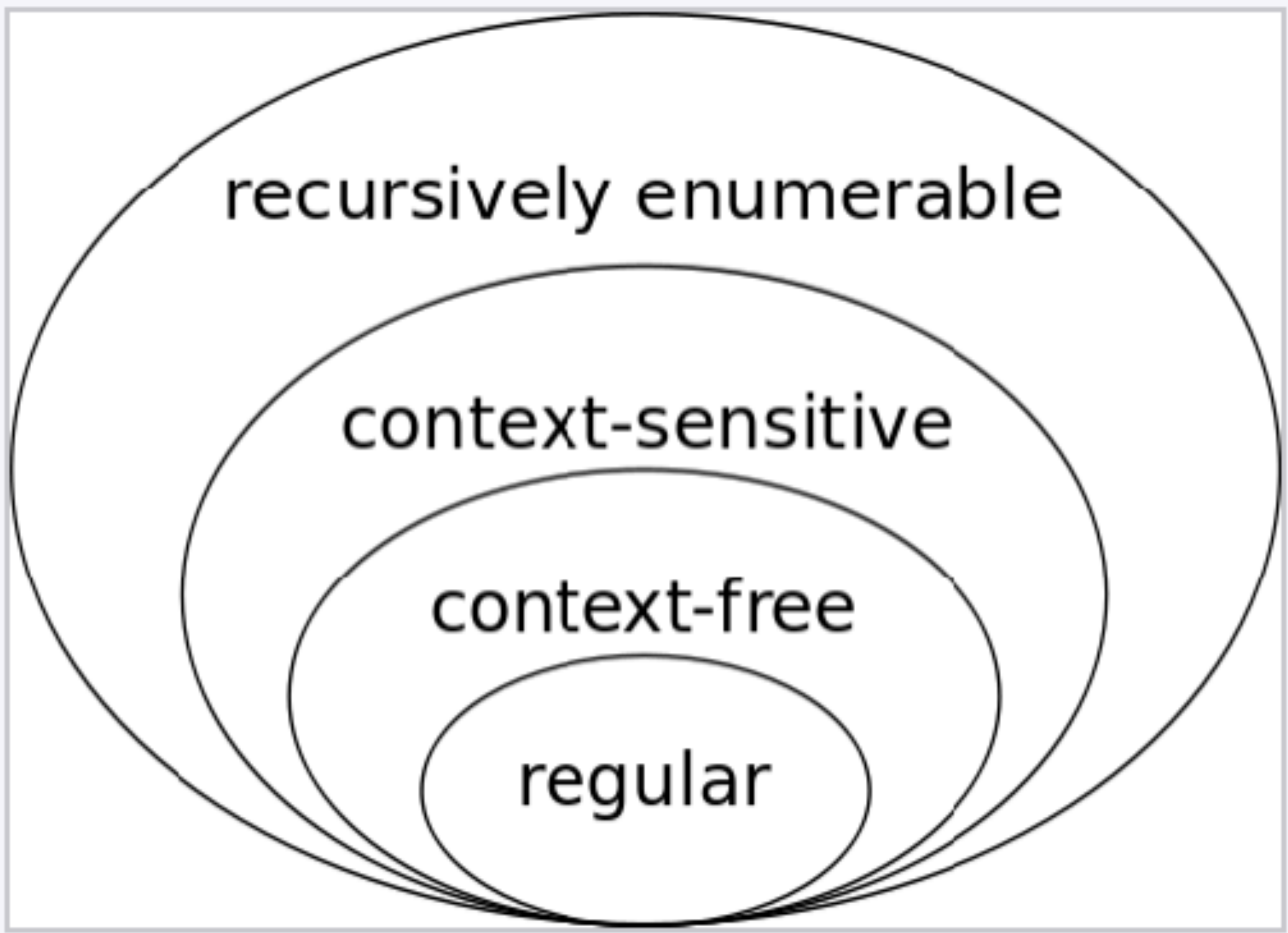
- Regular Expressions derive from the concept of a Regular Language;
- A regular language can be recognized by a finite automaton.
- Kleene’s theorem: regular expressions and regular language are equivalent.

# Regular Language

- Defined by “productions,” for example:

```
expr ::= term   { "+"   term   |   "-"   term }.
term ::= factor   {"*"   factor   |   "/"   factor}.
factor ::= floatingPointNumber   |   "("   expr   “)”.
```

# Chomsky Hierarchy




Regular language in classes of Chomsky hierarchy.

# Using regular expressions

- Regular expressions come in many different flavors (unfortunately):
  - Perl (Python?)
  - Java/POSIX (also used by other languages such as Scala)
- They are the basis of parsing (used for DSLs)
- Best resource for testing/understanding regex: https://regex101.com/ (but not Java-specific)
- RegexBuddy (but not free).

# Untyped parsing (Scala)

- OK, now we just need to be able to define the grammar that our parser can operate on:
  - Take a look at this set of "productions" in BNF (Backus-Naur form) followed by examples:

```
expr ::= term   { "+"   term   |   "-"   term }.
term ::= factor   {"*"   factor   |   "/"   factor}.
factor ::= floatingPointNumber   |   "("   expr   ")".
```

  - `1+4.5-3` is an *expr*; `2*3.14/5` is a *term;* `3.1415927` is a *factor*; `(7-5)` is also a *factor*.

Since 2.11, this is separate

  - The **Scala Parser Combinator** library allows us to code this parser with only a few substitutions:

"~" replaces " "; "rep(" replaces "{"; ")" replaces "}"; ";" replaces "." [although those ";" are entirely optional]

```
import scala.util.parsing.combinator._
class Arith extends JavaTokenParsers {
  def expr: Parser[Any] = term~rep("+"~term | "-"~term);
  def term: Parser[Any] = factor~rep("*"~factor | "/"~factor);
  def factor: Parser[Any] = floatingPointNumber | "("~expr~")";
}
```

each method defines a *Parser[Any]*

# Typed parsing

```
package edu.neu.coe.scala.parse
import scala.util.parsing.combinator._
class Arith extends JavaTokenParsers {
  trait Expression { def eval: Double }
  abstract class Factor extends Expression
  case class Expr(t: Term, ts: List[String~Term]) extends Expression {
    def term(t: String~Term): Double = t match {case "+"~x => x.eval; case "-"~x => -x.eval }
    def eval = ts.foldLeft(t.eval)(_ + term(_))
  }
  case class Term(f: Factor, fs: List[String~Factor]) extends Expression {
    def factor(t: String~Factor): Double = t match {case "*"~x => x.eval; case "/"~x => 1/x.eval }
    def eval = fs.foldLeft(f.eval)(_ * factor(_))
  }
  case class FloatingPoint(x: Any) extends Factor { def eval = x.toString.toDouble }
  case class Parentheses(e: Expr) extends Factor { def eval = e.eval }
  def expr: Parser[Expr] = term~rep("+"~term | "-"~term) ^^ { case t~r => r match {case x:
List[String~Term] => Expr(t,x)}}
  def term: Parser[Term] = factor~rep("*"~factor | "/"~factor) ^^ { case f~r => r match {case x:
List[String~Factor] => Term(f,x)}}
  def factor: Parser[Factor] = (floatingPointNumber | "("~expr~")") ^^ { case "("~e~")" => e match
{case x: Expr => Parentheses(x)}; case s => FloatingPoint(s) }
}
```