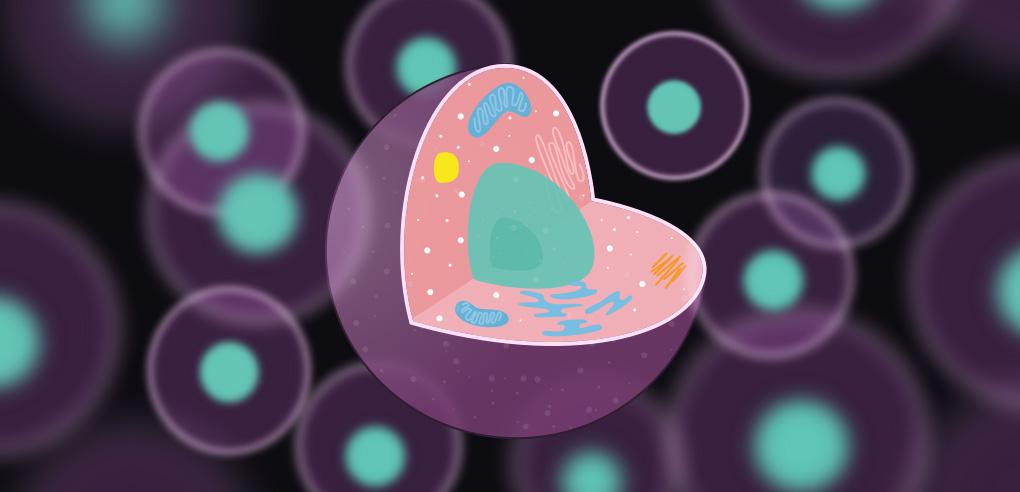
# Machines: A Language Acceptor Perspective

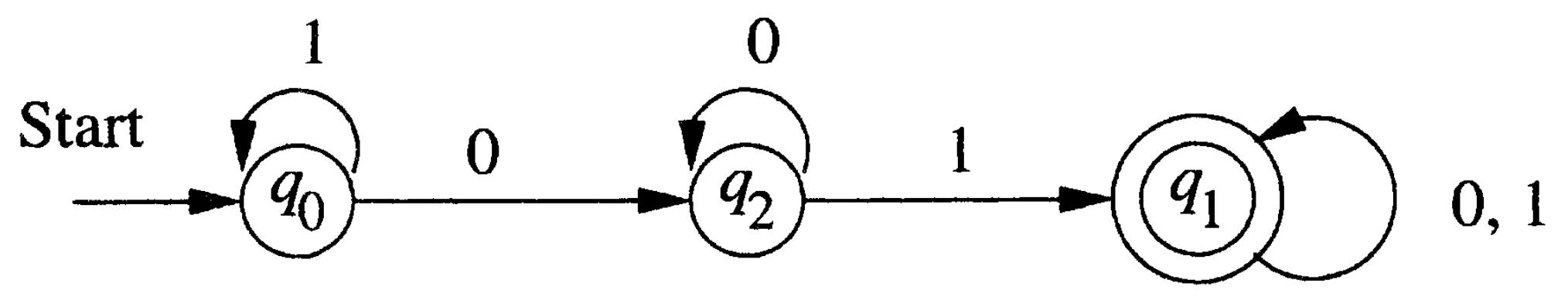
**\_\_**



## Finite Automata

In a sense finite automata are the “simplest” machines: based on the input, the finite automaton jumps from one state to another. Perhaps surprisingly, deterministic and nondeterministic (with or without epsilon-transitions) finite automata are equivalent, though in the worst case conversion from non deterministic to deterministic will require exponentially many states. Finite automata represent exactly **regular languages**, and another way to describe regular languages is through **regular expressions**.

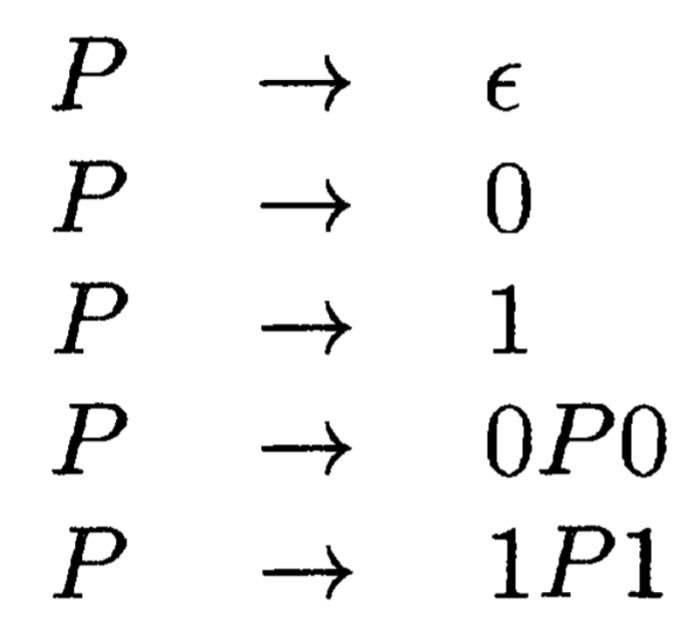
As an example, (deterministic) finite automata can accept all and only the strings of 0’s and 1’s that have the sequence 01 somewhere in the string.



## Pushdown Automata

Pushdown automata can be seen as finite automata augmented with a stack. This means the transition from one state to another now depends on the top element of the stack as well, in addition to the input symbol. Pushdown automata represent exactly **context-free languages**, and another way to describe context-free languages is using **context-free grammar**.

As an example, the language of palindromes is a context-free language, but not a regular language.

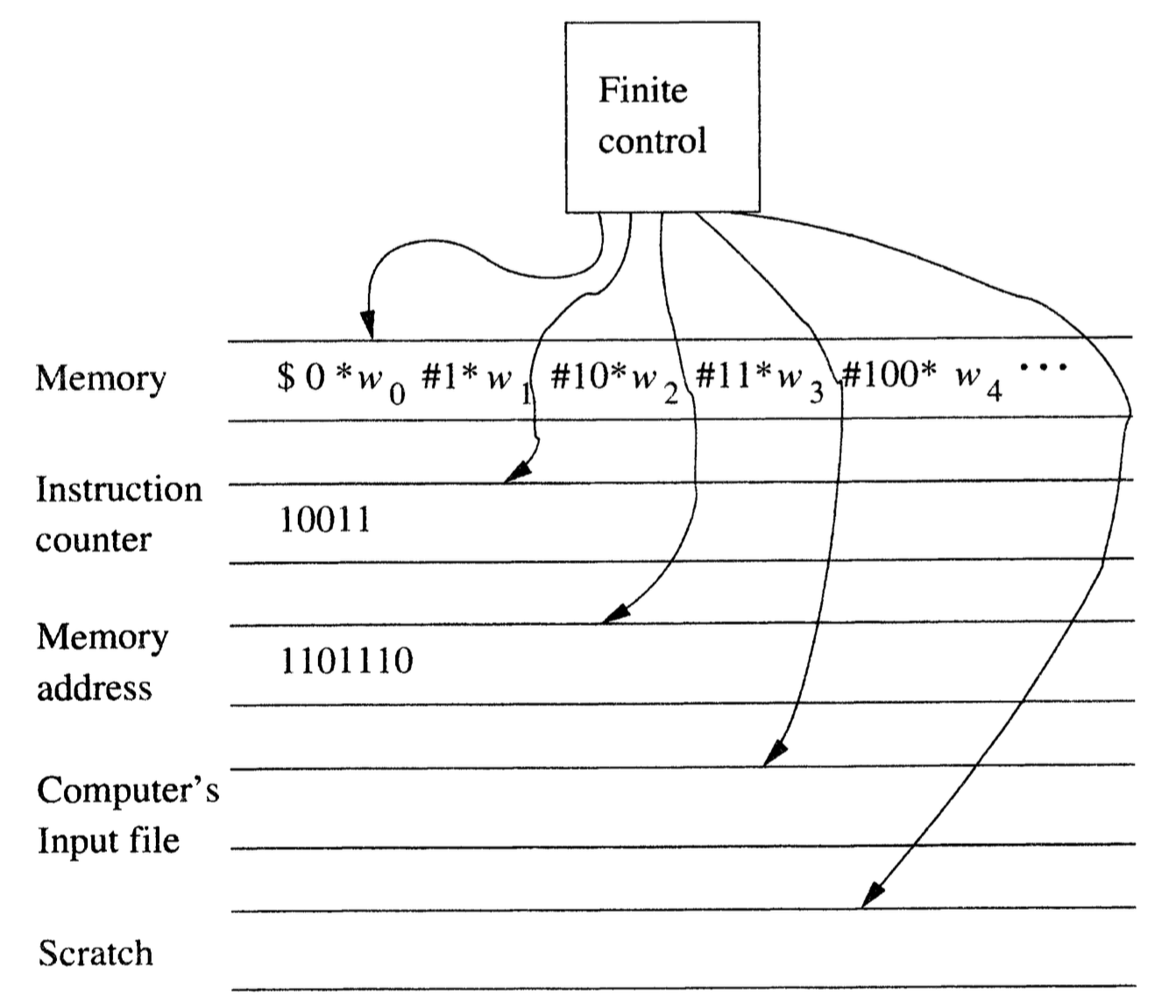


## Turing Machines and Computers

The definition of Turing machines is farther away from the definitions of finite automata and pushdown automata. It has a finite control that can move left or right on an infinite tape (initially holding the input string), while also having the ability to change the content on the tape. A lot of the variants are equivalent, including storage in the state, multiple tracks, multitape etc. Turing machines represent exactly **recursively enumerable languages**.

As an example, {anbncn| n1} is a recursively enumerable language, but not a context-free language.

While a Turing machine looks nothing like a PC, it has long been recognized as an accurate model for physical computing devices. I was very much convinced just by the following figure:



Therefore, the language for a Turing machine to accept is no different than the problem for a computer to solve. Obviously there are different difficulty levels, from hard to easy:

### No Turing Machines Can Accept

Despite the power of Turing machines, there are languages that no TMs can accept, i.e., they are outside the recursively enumerable language. In order to understand this, we need to develop codes for Turing machines, such that they can be numbered and ordered.

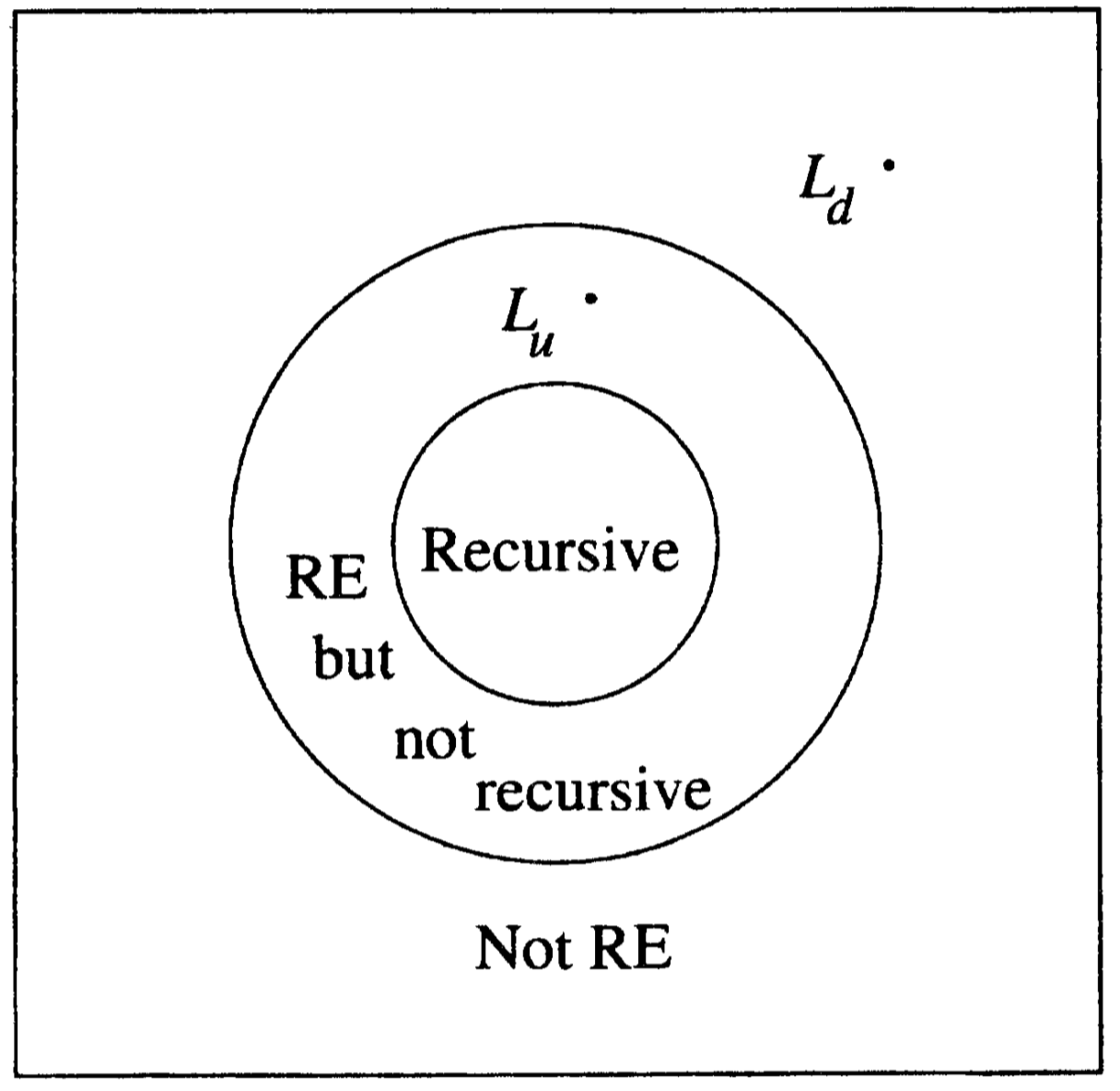
The idea is that: one rule in the TM’s transition function takes the form

*δ*(*q*,*Xj*)=(*qk*,*Xl*,*Dm*),where qi,Xj,qk,Xl,Dm are current state, current tape symbol, next state, rewrite tape symbol, move direction. We can encode this rule by the string

0*i*10*j*10*k*10*l*10*m* Suppose there are *n* n ordered rules in total, we can encode the entire TM with *C*111*C*211⋯*Cn*−111*Cn.* Now we can talk about the *i*-th Turing machine *Mi,* following the order *ϵ* is the first string, 0 is the second, 1 is the third, 00 the fourth, 01 the fifth, and so on.We are now ready to describe the **Diagonalization Language** *Ld* , the set of strings *wi* such that *wi* is not in *L*(*Mi*) . To understand this language, consider a table where row *i* column *j* is either 0 or 1, representing whether the TM *Mi* accepts input string *wj* . We may think of the *i*-th row as the characteristic vector for the language *L*(*Mi*) .The diagonal values tell whether *Mi* accepts *wi* . To construct *Ld* , we complement the diagonal. The trick is, this complement of the diagonal is itself a characteristic vector describing membership in some language, namely *Ld* . This characteristic vector disagrees in some columns with every row of the table. Thus, the complement of the diagonal cannot be the characteristic vector of any Turing machine!

### Turing Machine Not Guaranteed to Halt (Undecidable)

These are languages within RE, but are accepted in an inconvenient way: if the input is in the language, we will eventually know that, but if the input is not in the language, then the Turing machine may run forever, and we shall never be sure the input won’t be accepted eventually. These languages are called **not recursive**, and the corresponding problems are called **undecidable**.

The **Universal Language** *Lu* , the set of binary strings representing a TM and an input accepted by that TM, is an example of such. The code for the TM *M* and the code for the input string *w* are separated by string 111. The proof is by contradiction: if *Lu* were recursive, then its complement is also recursive by theorem. But if we have a TM to accept the complement of *Lu* , then we can construct a TM to accept *Ld* .

### Turing Machine Cannot Solve in Polynomial Time (Intractable)

Recursive languages (i.e., decidable problems) are nice, because they correspond to our informal notion of an “algorithm”, a well-defined sequence of steps that always finishes and produces an answer. However, there is still the issue of efficient versus inefficient.

Formally, we say a language is in the class NP (nondeterministic polynomial) if there is a *nondeterministic* TM that can accept all strings in the language within polynomial time of input length. One famous NP example is the Traveling Salesman Problem. The reason is that with a nondeterministic computer, we could guess a permutation of the nodes, and then compute the total weight for the cycle of nodes. No branch would use more than *O*(*n*) steps if the input was of length *n*. On a multitape NTM, we can guess a permutation in *O*(*n*2) steps and check its total weight in a similar amount of time.

### Turing Machine Solvable in Polynomial Time (Tractable)

We have finally reached the easiest level within RE languages. We say a language is in class P if there is a *deterministic* TM that can accept all strings in the language within polynomial time of input length.And we don’t know yet if P =NP !

###### 