How to Make Automata Examples and Definitions More Natural: From Explainable AI to Explainable Automata Theory

Ian Bautista Ambriz, Monet Nevarez Sanchez, Christian Servin,
Olga Kosheleva, Vladik Kreinovich, and Nguyen Hoang Phuong

Abstract An important part of Computer Science education is learning theoretical foundations of computing. This helps students better understand which problems are solvable and which are not, what tools they need to solve problems which are solvable, how to best implement these solutions – and also helps them to get a general understand how computers translate our instructions – i.e., our code – into efficient step-by-step computations. The corresponding Automata class is often difficult to students, because many concepts, ideas, algorithms, and proofs do not come naturally from the formulation of the corresponding problems: these ideas, algorithms, and proofs have come from the insights of founders of this field. This situation is similar to the need for make AI more explainable: we have interesting and often very effective ideas and results generated by AI, but it is difficult for us to under-

Ian Bautista Ambriz
Department of Computer Science, University of Texas at El Paso, 500 W. University
El Paso, Texas 79968, USA, e-mail: ibautistaambr@miners.utep.edu

Monet Nevarez Sanchez
Department of Computer Science, University of Texas at El Paso, 500 W. University
El Paso, Texas 79968, USA, e-mail: mnevarezsanch@miners.utep.edu

Christian Servin
Computer Science and Information Technology Systems Department, El Paso Community College,
919 Hunter, El Paso, TX 79915, USA, e-mail: cservin1@epcc.edu

Olga Kosheleva
Department of Teacher Education, University of Texas at El Paso, 500 W. University
El Paso, Texas 79968, USA, e-mail: olgak@utep.edu

Vladik Kreinovich
Department of Computer Science, University of Texas at El Paso, 500 W. University
El Paso, Texas 79968, USA, e-mail: vladik@utep.edu

Nguyen Hoang Phuong
Artificial Intelligence Division, Information Technology Faculty, Thang Long University
Nghiem Xuan Yem Road, Hoang Mai District, Hanoi, Vietnam
e-mail: nhphuong2008@gmail.com

stand them – since AI usually does not provide us with an explanation of how it came up with these ideas. In this paper, on the example of several basic Automata concepts and proofs, we show that the use of invariances can help to make many such ideas more natural.

1 Introduction

Many ideas and methods that we teach our students come from a spark of genius. In computer science – as well as in many other disciplines – researchers and practitioners have come up with many creative ideas and techniques. Computing examples are (see, e.g., [1]):

- how to sort numbers faster that we would usually do, how to multiply numbers faster than what we learn in school, how to multiply matrices faster that we learn in most matrix algebra classes,
- how to create a safe encoding that is difficult to break,
- how to prove that it is not possible to predict when a program will halt, etc.

In each such case, coming up with the corresponding method was a stroke of genius.

This same spark of genius makes it difficult for students to learn these ideas and methods. We use these methods, we proudly teach them to students – because they need to use them. But these methods are not easy for students to learn – because they do not naturally follow from the problem, each of them looks like an alien trick.

To students, this sometimes start feeling like Happy Potter's Hogwarts school, where students have to memorize incantations like Abracadabra without any understanding of how to explain their work. Similarly, students have to memorize many such tricks.

How can we make this material easier for students? In many cases, it later turns out that it is possible to explain such a method – after the fact – in a more natural way.

This is similar to the need for explainable AI. The need to make the material more understandable is similar to one of the main challenges of modern AI: how to make AI results more understandable. In both cases – when trying to understand AI results and when trying to learn Automata material – we have complex difficult-to-understand-why ideas generated by a "black box": AI's "brain" or a human brain. In both cases, we want to better understand where these idea came from, and how to make their appearance more natural.

What we do in this paper. In this paper, we show how to do it – on the example of an Automata course; see, e.g., [3].

2 Our main idea: use invariances

What are invariances? To understand our main idea, let us recall how we gain new knowledge about the world. How do we know that if we drop a pen, it will start falling down with the acceleration of 9.81 m/sec^2? Someone (actually Galileo) dropped a pen at some place and measured the acceleration. Then, he (and others) repeated this experiment at different locations and at different moments of time – and observed the same behavior. In other words, if we move to a new location or move ahead in time, the result does not change. We can this say that this result is *invariant* with respect to shifts in space and in time.

One may say that we believe that the pen will fall down because it follows from the Newton's law of gravity. But how do we know that the Newton's law of gravity holds? Newton observed it when he analyzed why an apple fell on his head, he observed it when he analyzed the motion of the Moon and of planets – and he observed that the law does not change if we shift in space or in time.

Invariance is behind all physical laws. Invariances – that physicists call *symmetries* – are one of the main tools of modern physics; see, e.g., [2, 4]. It is known that invariances not only help us understand the physical world, they can also help us to select the best alternatives. Let us explain how this works.

What do we mean by "the best alternative"? To explain the relation between invariances and the search for the best alternative, let us recall what we mean by "the best alternative".

Suppose that we need to select between several alternatives. Let us denote the set of all these alternatives by A. Out of these alternatives, we want to select the one which is, in some sense, the best. By "the best", we mean the alternative that is better than – or at least of the same quality – as all others. Let us denote:

- "a is better than b" by $a > b$, and
- "a is of the same quality" by b by $a \sim b$.

In these terms, an alternative a_{opt} is optimal of for all alternatives a, we have

$$\text{either } a_{\text{opt}} > a \text{ or } a_{\text{opt}} \sim a.$$

What if we have several best alternatives? If there are several optimal alternatives, then we can use this non-uniqueness to optimize something else.

For example, if we are selecting a sorting algorithm for sorting n values, and we want an algorithm with the smallest average time complexity, then we have several algorithms with the same average time proportional to $n \cdot \log(n)$: mergesort, heapsort, quicksort, etc.; see, e.g., [1]. Since there are several alternatives that are equally good with respect to this criterion, we can select, among them, the algorithm(s) that have the smallest worst-case complexity: that would leave mergesort and heapsort. We can again use the remaining non-uniqueness to select, e.g., the algorithm that requires the smallest amount of memory – which leaves only the heapsort.

Similarly, if we are selecting a candidate for a faculty position, and we have two candidates with equally good research records, a reasonble idea is to select the one with the best teaching skills.

In general, we can continue to utilize additional criteria until we end up with a single alternative that is the best according to all of these criteria.

How invariances can help: general idea. On the set of all alternatives, there are some possible transformations g, g', etc. that should not affect the relative quality of the alternatives:

- if $a > b$, then after applying any of these transformations g, we should still have $g(a) > g(b)$, and
- if $a \sim b$, we should have $g(a) \sim g(b)$.

Suppose now that we have a criterion $(<, \sim)$ that is:

- final – i.e., there is exactly one optimal alternative, and
- invariant with respect to some transformations g – i.e.,
  - $a > b$ implies $g(a) > g(b)$, and
  - $a \sim b$ implies $g(a) \sim g(b)$.

Let us show that in this case, the optimal alternative a_{opt} does not change under all such transformation f, i.e.,

$$g(a_{\text{opt}}) = a_{\text{opt}}.$$

How invariances help: an example. Before we explain why the optimal alternative should itself be invariant, let us provide a simple example. Suppose that we are looking for a highest point on some structure that is invariant with respect to rotations around its axis – e.g., an inverted cone, or a cylinder base followed by a conic top.

Let us also assume that there is exactly one highest point. Then this point must be on located on the axis.

If the highest point H was not located on the axis, all the points obtained from H by rotation around this axis would have the same largest height. This contradicts our original assumption that H is the *only* highest point. So, the only way for the point H to be the highest point is it is located in the axis.

How invariances help: a proof. Let us show how we can prove the above statement about the relation between optimality and invariance. The fact that a_{opt} is optimal means that for every alternative a, we have

$$a_{\text{opt}} > a \text{ or } a_{\text{opt}} \sim a.$$

In particular, if we apply the inverse transformation g^{-1} to a, we get

$$a_{\text{opt}} > g^{-1}(a) \text{ or } a_{\text{opt}} \sim g^{-1}(a).$$

Since the criterion is invariant, this implies that

$$g(a_{\text{opt}}) > g(g^{-1}(a)) = a$$

or

$$g(a_{\text{opt}}) \sim g(g^{-1}(a)) = a.$$

So, for every a, we have

$$g(a_{\text{opt}}) > a \text{ or } g(a_{\text{opt}}) >\sim a.$$

By definition of optimality, this means that the alternative $g(a_{\text{opt}})$ is optimal. But there is only one optimal alternative, so

$$g(a_{\text{opt}}) = a_{\text{opt}}.$$

Conlusion. So, to find the optimal alternative, we can look for alternatives that are invariant with respect to the corresponding transformations.

3 How to find a language which is not regular

Let us describe the first example where an explanation is needed.

How a typical Automata class starts. The Automata class usually starts with finite automata. To describe a finite automaton, we need to list:

- a finite set Q whose elements are called *states*; one of these states q_0 is called a *starting state*;
- a subset F of the set of states; its elements are called *final states*, or, alternatively, *accept states*;
- a finite set S who elements are called *symbols*; and
- a transition function δ that assign, to each state q and each symbol s, the next state $\delta(q,s)$.

The input to the automaton is a *word*, i.e., a sequence of symbols. We start in the state q_0. We read the symbols from the word one by one. Each time we read a symbol, we use the transition function to decide what will be the next state. If after reading all the symbols we end up in an accept state, we say that the word is *accepted* by the automaton; otherwise we say that the word is *rejected* by this automaton.

Let us give a simple example of an automaton that takes numbers, i.e., sequences of digits, and accepts even numbers, i.e., numbers that end in an even digit (0, 2, 4, 6, or 8). This automaton has three states:

- the start state q_0 meaning that we have not read any digits yet;
- the state e meaning that the last read digit was even; this is the only final state; and
- the state o meaning that the last read digit was odd.

In this case, $F = \{e\}$.

The set of symbols is the set of all digits: $S = \{0, 1, \ldots, 9\}$. In each state, when we see an even digit, we go to the state e, and when we see an odd digits, we go to the state o. So, to check that 16 is an even number, this automaton does the following:

- it starts at the state q_0;
- then, it reads the first symbol of the word, which is 1, and moves to the state o;

- finally, it reads the second symbol of the word, which is 6, and moves to the state e.

We have read all the symbols and we are in a final state, so the word 16 is accepted by this automaton.

For each finite automaton, we can form the set of all the words that are accepted by this automaton. In theory of computation, sets of words – that we will denote by A, B, ... – are called *languages*. The language corresponding to an automaton is called *regular*.

In an Automata course, it is usually shown that regular languages are exactly the ones that can be described by so-called *regular expression*. A regular expression is any expression that can be obtained from letters by using:

- union $A \cup B$, the set of all the elements that are either in A or in B or in both; for example, if $A = \{\text{cat}, \text{dog}\}$ and $B = \{\text{dog}, \text{wolf}\}$, then

$$A \cup B = \{\text{cat}, \text{dog}, \text{wolf}\};$$

- concatenation $AB = \{ab : a \in A \,\&\, b \in B\}$, the set of all concatenations ab of words $a \in A$ and $b \in B$ (concatenation means the word a followed by the word b; for example, if a = "blue" and b = "cat", then ab = "bluecat"); so for $A = \{\text{red}, \text{blue}\}$ and $B = \{\text{cat}, \text{dog}\}$,

$$AB = \{\text{redcat}, \text{reddog}, \text{bluecat}, \text{bluedog}\};$$

- so-called Kleene star $A^* = \Lambda \cup A \cup AA \cup AAA \cup \ldots$, where Λ denotes the empty string; for example, for $A = \{\text{cat}\}$,

$$A^* = \{\Lambda, \text{cat}, \text{catcat}, \text{catcatcat}, \ldots\}.$$

At first, in an Automata course, we have constructive and positive results, where students are given explicit algorithms:

- for describing a non-deterministic automaton based on a regular expression,
- for transforming a non-deterministic finite automaton into a deterministic one,
- for showing how to transform an automaton into a regular language, etc.

All these algorithms are straightforward and reasonably natural, they do not use any unexpected tricks.

And then comes the first theoretical result – that some languages are not regular. It is usually proved on the example of the language

$$\{a^n b^n, n = 0, 1, 2, \ldots\} = \{\Lambda, ab, aabb, aaabbb, \ldots\},$$

where Λ denotes the empty string, and a^n means letter a repeated n times. In words from this language, first a letter a is repeated several (n) times and then another letter b is repeated the same number of times.

There is no motivation neither for this example nor for the related proof. To the students, it come as a result of a stroke of genius. This may help them appreciate

the smartness of the founders of theory of computation, but it does not help them remember the proof and its main ideas.

Comment. In general, it is always much easier to remember the material in which one part naturally leads to another one than the material in which students need to remember a significant number of unnatural tricks.

What we plan to do about this problem. In this section, we will show that appropriate invariance ideas can lead to natural explanation of why this particular language was selected.

How we plan to do it. In a nutshell, what we would like to do is to find the most natural of all non-regular languages. As we have mentioned in the first section, if we have a final optimality criterion which is invariant with respect to some natural transformations, then the optimal alternative is itself invariant with respect to these transformations. To use this idea, let us analyze which are reasonable transformations here, i.e., reasonable transformation that acts on words (i.e., strings) and/or on segments of these words.

What are reasonable local transformations. One of the most natural local transformations is a shift. The simplest of the shifts is by one step. It can be one step to the right or one step to the left. Without losing generality, let us consider a shift one step to the right.

What do we mean by saying that a word is locally invariant with respect to such a shift? It means that if we originally saw some letter, and then we shift one step to the right, we will see the exact same letter.

Which words are invariant with respect to local transformations? How many such invariances can we have in a word of length n? In principle, we can have $n-1$ such invariance:

- for the first letter – meaning that the second letter is the same as the first one;
- for the second letter – meaning that the third letter is the same as the second one;
- $\ldots$; and
- for the second-to-last $(n-1)$-st letter – meaning that the last (n-th) letter is the same as the $(n-1)$-st one.

Words with the maximum number of local transformations do not form a non-regular language. We are looking for the maximally invariant word, i.e., for the word that is invariant with respect to the largest possible number of invariances. So let us look for a word that has all such invariances. We thus look for the word in which the second letter is the same as the first one, the third letter is the same as the second one, etc. Thus, all the letters in this word are the same, i.e., the word has the form $a^n = a \ldots a$ for some letter a.

However, this does not lead us to a language which is not regular. The set $\{a^n : n = 0, 1, \ldots\}$ of all such words *is* regular. Namely, it is the result a^* of applying the Kleene star operation A^* to a language that consists of a single 1-letter word a.

Limiting ourselves to only local transformations does not lead to a non-regular language. Since we cannot have all $n-1$ possible local invariances, let us look for the words that have one fewer local invariance, i.e., that have $n-2$ invariances. This means that:

- in all but one cases, the next letter is the same as the previous letter, but
- in one of the cases, the next letter is different from the previous letter.

In other words:

- at first, we have several repetitions of the same letter,
- then we get another letter and after that, we have repetitions of this new letter.

If we denote the first repeating letter by a and the second repeating letter by b, then we get a word of the type $a\ldots ab\ldots b$. In general, the set of all such words $\{a^n b^m : m,n = 0,1,\ldots\}$ can be described by a regular expression a^*b^*. We can have $n-3$ invariances – and still get a regular expression: in this case $a\ldots ab\ldots bc\ldots c$ which is $a^*b^*c^*$. Similarly, we get a regular expression if we limit ourselves to $n-4$ local invariances, etc.

So, it looks like we cannot get a non-regular language if we only use local invariances. To get a non-regular language, we need to also add some global invariance(s) to the local ones.

What global transformations are possible and which words are invariant with respect to these transformations. The first natural global transformation is related to the fact that the language remains regular or not if we simply rename the variables. For the above case of words formed by two letters a and b, the only possible renaming is replacing each a with b and each b with a.

Can there be a word that is invariant with respect to this renaming? Not really, since if we change a at any position to b, the word will change. So, to find a word that is invariant with respect to global transformations, we need to consider other global transformations as well. One natural global transformation is word reversal, when you form a new word by listing all the symbols of the old word in the reverse order: e.g., *cat* becomes *tac*. Based on the definition of a finite automata, it may not be clear why if we reverse all the words in a regular language, we will still get a regular language. However, this statement becomes clear if we use the definition of a regular language – a language that can be obtained from letters by using union, concatenation $AB = \{ab : a \in A \,\&\, b \in B\}$ and Kleene star $A^* = \Lambda \cup A \cup AA \cup AAA \cup \ldots$. In this case, if we reverse the regular expression – for example, replace $(a \cup b)^* b$ with $b(b \cup a)^*$ – we get exactly the reversal of the original language.

No locally invariant words – e.g., words of the type $a^n b^m$ – are invariant with respect to such a reversal. Indeed, each such word starts with the letter a and ends in a letter b, while a reversed word starts with the letter b and end in a letter a, so the original word and the reversed word are always different.

What can we do? We have two global transformations. As we have mentioned earlier, it is also reasonable to consider a composition of two transformations, e.g., when we first rename and then reverse. Which words of the type $a^n b^m$ are invariant with respect to this composition?

- first, we rename the letters; this brings us from the original word $a^n b^m$ to the renames word $b^n a^m$; and
- then, we reverse the resulting word $b^n a^m$, making it $a^m b^n$.

Invariance means that the final word $a^m b^n$ is the same as the original word $a^n b^m$. This is happens when $n = m$, so we end up with words of the type $a^n b^n$.

Resulting explanation. To find an example of a non-regular language, we take the set of all the words that:

- have the largest number of local invariances that do not lead to a regular language, and
- are invariant with respect to a global invariance – a composition of renaming and reversal.

This provides a natural explanation of the choice of this word.

4 How to prove that this language is not regular

How this is proved now. The usual proof is based on analyzing the *trajectory* of the automaton, i.e., on analyzing the sequence of states through which the automaton passes as it processes the given word symbol-by-symbol. Specifically, the proof uses the so-called Pumping Lemma, according to which each word of sufficient length accepted by a finite automaton can be divided into three consecutive parts – by finding the first repeating state in the trajectory of the automaton that recognizes this word.

A natural question. How can we make this idea natural? How can we explain, to the students, why are we selecting repeating states?

Towards an invariance-based explanation. In the above explanation of selecting a word, we used a very specific local transformation: going from one symbol to the next one. In selecting the word, we checked when the word is invariant with respect to this transformation. Now, in addition to the *letters* that form the word, we also talk about *states* that the automaton passes through when it reads the letters of the woird one by one. This sequence of states is known as the *trajectory*. So, in addition to the shift-by-one transformation of the word, it makes sense to consider a similar shift-by-one transformation for the trajectory.

It is rare that the actual trajectory is invariant under such a transformation – it would mean that we have two repeating states one after another, and many finite automata do not have any transitions from a state back to itself. To consider all possible finite automata, we this need to consider more general local transformations. For example, instead of a shift by one step we can consider shifts by several steps. For such transformations, invariance means that two states repeat. So, from the invariance, viewpoint, it makes sense to consider repeating pairs of states – and it is natural to consider the first such pair that appears as the finite automaton processes the given word.

This explains why we consider the first repeating pair of states. Once we have this pair, the original word w can be naturally divided into the following three parts:

- the part before the first repeating state; this part is usually denoted by x;
- the part between the two repeating states; this part if usually denoted by y; and
- the part after the second repeating state; this part is usually denoted by z.

Out of these three parts, the part y has some symmetry – it starts and ends at the same state. In symmetry terms, this part is invariant with respect to swapping the starting and ending states of this part.

The original word is thus equal to the concatenation of these three parts: $w = xyz$. In the expression xyz, there are no local symmetries. To add a local symmetry, we need to repeat one of the three letters that form this word. From the symmetry viewpoint, let us repeat the part that has some symmetry – i.e., the part y. This way, we get the word $xyyz$.

The Pumping Lemma shows that the word $xyyz$ also belongs to the original language.

We can also show that both parts x and y are among the first p symbols of the word, where p is the number of states in the correpsonding finite automaton. Indeed, when the automaton reads the first p letters, it goes through $p+1$ states:

- the starting state, before we read any letters, and
- p states that we get after reading each of the p letters.

Since there are only p different states, some of these $p+1$ states must repeat. So, the first repeating pair of states must occur when we are still in the first p letters. Thus, every word from the language whose length is at least p can be described as a concatenation xyz, where both parts x and y are within the first p letters of the word.

How this helps to prove that the language $\{a^n b^n, n = 0, 1, \ldots\}$ is not regular: reminder. The Pumping Lemma can help to prove that the language $\{a^n b^n, n = 0, 1, \ldots\}$ is not regular.

Indeed, suppose that this language is regular, and let p denote the number of states in the corresponding finite automaton. Let us take the word $a^p b^p$, in which first the letter a is repeated p times, and then the letter b is repeated p times. The length of this word is larger than p, so the Pumping Lemma applies. For the selected word, the first p letters are all a's. Thus, the part y consists of only a's. So, if we go from the word xyz to the word $xyyz$, we add a's – but we do not add any b's. In the original word $xyz = a^p b^p$, we had equal number of a's and b's. Now that we added some a's, we have more a's than b's. This means that the word $xyyz$ does not belong to our language – but by Pumping Lemma, it should. This contradiction proves that our assumption – that this language is regular – is false. So, the language $\{a^n b^n, n = 0, 1, \ldots\}$ is not regular.

5 Other examples of non-regular languages

What we do in this section. In this section, we show that several other examples of non-regular languages that are usually taught in the Automata class can also be explain by appropriate symmetries.

The language of all the words that have equal number of a's and b's. For words from this language, a numerical characteristic – namely, the number of letters of a given type – is invariant with respect to swapping the letters a and b. This language is also not regular.

To prove that this language is not regular, we need to select a word from this language for which the Pumping Lemma will lead to a contradiction. In view of the general symmetry approach, the most appropriate word should be the one with the largest number of symmetries. So, using the same arguments as before, we end up with the same word $a^n b^n$ – for which indeed we get a contradiction. **The language of all the words that have equal number of a's and b's.** For words from this language, a numerical characteristic – namely, the number of letters of a given type – is invariant with respect to swapping the letters a and b. This language is also not regular.

To prove that this language is not regular, we need to select a word from this language for which the Pumping Lemma will lead to a contradiction. In view of the general symmetry approach, the most appropriate word should be the one with the largest number of symmetries. So, using the same arguments as before, we end up with the same word $a^n b^n$ – for which indeed we get a contradiction.

The language of palindromes. One of the global symmetries that we have used before is the word reversal. A natural question is: what if we consider this symmetry on its own – i.e., consider all the words that are invariant under reversal? In plain English, this means that the word reads the same when we read it from the beginning to the end and from the end to the beginning, e.g., the word *abba*. Such words are known as *palindromes*. It is known that the language of all palindromes is also not regular.

To prove that this language is not regular, we need to select a word from this language for which the Pumping Lemma will lead to a contradiction. In view of the general symmetry approach, the most appropriate word should be the one with the largest number of symmetries. Similar to the main example:

- if we have the largest possible number of local shift-by-one invariances, namely, $n-1$ such invariances, then we end up with words of the type a^n – and, as we have mentioned earlier, the set of all such words is regular;
- if we have the next largest number of local shift-by-one invariance, namely, $n-2$ such invariances, then we end up with words in which the letter changes once, i.e., words of the type $a^n b^m$, but these words are not palindromes.

Since we cannot have $n-2$ local invariances, let us try $n-3$ such invariances, so that the letter charges twice. In this case, we have:

- either words in which the second change leads to a yet different letter, i.e., words of the type $a^n b^m c^p$ – which are not palindromes
- or the words in which the second change brings us back to the same letter, i.e., words of the type $a^n b^m a^p$.

Not all words of the type $a^n b^m a^p$ are palindromes. For such a word to be a palindrome, we need to have $p = n$. Thus, we arive at the set of palindrome words of the type $a^n b^m a^n$. In particular, for the word $a^p b^m a^p$, the Pumping Lemma leads to the word $xyyz = a^{p+t} b^m a^p$ for some $t > 0$ – the word which is not a palindrome. This proves that the language of palindromes is not regular.

6 How to find a language which is not context-free

Let us analyze what happens once we prove that some languages are not regular: need to add stacks. In a usual Automata class, once we prove that there exists a language that is not regular – i.e., for which we cannot use a finite automaton to decide whether a given word belongs to this language – a natural idea is to think what can be added to the finite automaton so that with this addition, it will be able to recognize words like $a^n b^n$. In principle, we can add many different data types to a finite automaton – a linked list, an array, a 2-D array, etc. Since we are talking computations, a natural idea is to add a data type that is the most natural for computers, that does not require a complex design.

This may not be clear when we study data types, but the most natural data type for a computer is a *stack*. For example, when you forget to add a stopping criterion to a recursive program, the error message you get is "Stack overflow". This message sometimes surprises students, since they did not use any stacks in their programs – but the answer to this surprise is that this is not about a stack designed in a program, it is about the computer memory that it actually organized as a stack – Last In, First Out.

The idea of a pushdown automaton. A finite automaton equipped with a stack forms what is known as a *pushdown automaton*. Let us explain, in some detail, how it works. Let us recall that in a finite automaton, when in a state q we see a symbol s, we just change the state to the new state q'. This new state $\delta(q,s)$ is uniquely determined by the state q and the symbol s. So, the corresponding rules have the form $q \xrightarrow{s} q'$.

In a pushdown automaton, in addition to the nest symbol s of the analyzed word, we can also read the top symbol t from the stack – which symbol is then popped from the stack – and we can also push some symbol p into the stack. The general rules have the form $q \xrightarrow{s,t \to p} q'$. In this case, the next state q' depends not only on the initial state q and on the symbol s, it can also depend on what is located on top of the stack.

Popping and pushing are not always necessary. To describe that in a particular transition we do not want popping, we use the special symbol ε instead of t. This

symbol is familiar to students from calculus, where it is used in the definition of the limit – to describe a small number (so small that in the limit it is not felt at all). Similarly, if we do not want to push anything into the stack, we use the same "negligible" symbol ε instead of p.

Context-free grammars: an alternative way to describe the same concept. It is known that languages recognized by pushdown autonata can be described in different way: as so-called *context-free grammars*. The notion of a context-free grammar comes from the usual way of describing programming languages. For example, an if-then statement in Java has two forms: "if (condition) statement" or "if (condition) statement else statement". In this description:

- the symbols i and f (that form the word "if"), (,), e, l, s, and e will appear in each actual conditional statement; such symbols are called *terminal* and are usually described by small letters, while
- the terms "condition" and "statement" need to be replaced by the actual condition or statement; such terms are called *variables*; they are usually described by capital letters, e.g., C for condition and S for statement.

If we add the symbol I to describe if-then statements, then the above rules take the following form: $I \to if\,(C)\,S$ and $I \to if\,(C)\,S\,else\,S$. Here, we can replace C with any condition, irrespective of the context, i.e., irrespective of what is before or after this symbols; this is what this construction is known as a *context-free grammar*.

In general, we have a finite set of terminal symbols, a finite set of variables – one of which (e.g., S_0) is designated as the starting variable – and we have rules of the type $V \to w$, where V is a variable, and w is a word composed of terminal symbols and/or variables. Applying this rule means we can replace one of the occurrences of the variable V in any word with the word w. We say that a word consisting of terminal symbol can be generated by the context-free grammar if we can get this word starting with the starting variable S_0 and applying rules from this grammar.

Not all languages are context-free. After showing that many important languages are indeed context-free, a natural question appears: are all languages context-free? It turns out that some languages are *not* context-free. The usual example of such a language is the following language:

$$\{a^n b^n c^n, n = 0, 1, 2, \ldots\} = \{\Lambda, abc, aabbcc, aaabbbccc, \ldots\}.$$

Natural question. How can we explain the choice of this language? Let us see how we can use symmetries to explain this choice.

Let us start – as before – with using local symmetries. In the example of a non-regular language, we use words that have $n-2$ local invariances.

A seemingly natural idea is to use the same number of local invariance again. However, this does not work: as we have mentioned, words that are invariant with respect to all such transformations have the form $a^n b^n$. While the set $\{a^n b^n, n = 0, 1, 2, \ldots\}$ of all such words is not regular, it is known to be context-free.

Since we cannot find a context-free language by using $n-2$ local invariances, let us take one fewer invariances, i.e., $n-3$. In this case, as one can check, we have words of the type $a^n b^m c^p$ for some natural numbers n, m, and p.

The language of all such words is context free, it is even regular: it is $a^* b^* c^*$. To come up with a language which is not context-free, let us add invariance with respect to the global transformation that we considered in Section 2: when we rename the variables and reverse the word. Each word $a^n b^m c^p$ starts with a and ends in c. If we reverse it, we get the word $c^p b^m a^n$ that starts with c and ends with a. So, to make the result the same, we need to swap letters a and c. Once we swap and reverse, we get the word $a^p b^m c^n$. This returns the same word if $p = n$, i.e., if we consider words of the type $a^n b^m c^n$. This language is not regular, but it is context-free.

Indeed, we can form a grammar that generates exactly all the words from this language. In this language we have a starting variable S, and the rules $B \to \varepsilon$, $B \to Bb$, $S \to B$, and $S \to aSc$. Here, B leads to all the words consisting of only letter b, and then S corresponds to the words in which we add, to an only-b-word, equal number of a's to the left and c's to the right. This is exactly all the words of the type $a^n b^m c^n$.

To form a language that is not context-free, a natural idea is to add symmetries that are kind of intermediate between local and global symmetries. Namely, instead of applying the global symmetry that we previously used – reversal followed by swapping the letters – to the original word, let us try to apply it to some reasonably selected part of the word $a^n b^m c^n$.

It is reasonable to consider the parts consisting of the whole a, b, and c blocks. If we only consider one of the blocks a^n, b^m, and c^n, then each of these blocks is already maximally symmetric with respect to local symmetries – we cannot get anything new by adding intermediate symmetries. So, let us consider parts consisting of two blocks, i.e., the parts $a^n b^m$ or $b^m c^n$.

- For the first of these parts $a^n b^m$, reversal leads to $b^m a^n$. Then swapping the two letters a and b that form this part leads to $a^m b^n$. The requirement that the part be invariant under this transformation means that the transformed part $a^m b^n$ should be the same as the original part $a^n b^m$. This means that we should have $m = n$, i.e., this part should have the form $a^n b^n$. With such a part, the whole world $a^n b^m c^n$ takes the form $a^n b^n c^n$.
- For the second of these parts $b^m c^n$, reversal leads to $c^n b^m$. Then swapping the two letters b and c that form this part leads to $b^n c^m$. The requirement that the part be invariant under this transformation means that the transformed part $b^n c^m$ should be the same as the original part $b^m c^n$. This means that we should have $m = n$, i.e., this part should have the form $b^n c^n$. With such a part, the whole world $a^n b^m c^n$ takes the form $a^n b^n c^n$.

So, in both cases, we get the word $a^n b^n c^n$ – which is exactly the word used in the usual proof that the language of all such words is not context-free.

Remaining open problem. In the non-regular example, not only we showed that the usual example of a non-regular language can be obtained by postulating natural

invariances. We also showed that the Pumping Lemma – that is used to prove that this language is not regular – can also be motivated by invariances.

In the non-context-free case, so far, we only found out how to explain the usual example in terms of invariances. We expect that it should be possible to provide the motivation for the proof this way too, but so far, we have not yet found out how to do it. It would be nice if someone can help with this.

Acknowledgments

This work was supported by the AT&T Fellowship in Information Technology, by the Institute for Risk and Reliability, Leibniz Universitaet Hannover, Germany, by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) Focus Program SPP 100+ 2388, Grant Nr. 501624329, by the European Union under the project ROBOPROX (No. CZ.02.01.01/00/22 008/0004590), by the Center of Excellence in Econometrics, Faculty of Economics, Chiang Mai University, Thailand, by the Ho Chi Minh City University of Banking, Vietnam, and by Thang Long University, Hanoi, Vietnam.

References

1. Th. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press, Cambridge, Massachusetts, 2022.
2. R. Feynman, R. Leighton, and M. Sands, *The Feynman Lectures on Physics*, Addison Wesley, Boston, Massachusetts, 2005.
3. M. Sipser, *Introduction to the Theory of Computation*, 3rd Edition, Course Technology Inc., Boston, Massachusetts, 2021.
4. K. S. Thorne and R. D. Blandford, *Modern Classical Physics: Optics, Fluids, Plasmas, Elasticity, Relativity, and Statistical Physics*, Princeton University Press, Princeton, New Jersey, 2021.