1 – Introduction to Formal Languages

Luc Lapointe
luc.lapointe@ens-paris-saclay.fr
home.lmf.cnrs.fr/LucLapointe/

Abstract
Formal Languages is a mathematically rigorous framework used to define "problems" we want our computers to solve, and to prove that some problems are out of reach of even our more powerful supercomputers. It also introduces *grammars*, a convenient tool to describe existing or new programming languages, including quantum ones.

Introduction activity: Towers of Hanoi

The "Towers of Hanoi" is a mathematical puzzle invented in 1883 by French mathematician Édouard Lucas, first presented as a game discovered during a trip.

It consists of three rods (left, middle and right) and *n* disks of increasing diameter, with a hole in the middle of each to slide onto any rod. The **starting position** of the puzzle is all disks stacked on the left rod by increasing diameter order, the smallest at the top. The **goal** is to move all disks to the right rod by obeying the following **rules**:
- You can only move disks one by one;
- You can only move a disk from the top of a stack to the top of another stack or to an empty rod;
- You can not move a disk on top of another *smaller* disk.

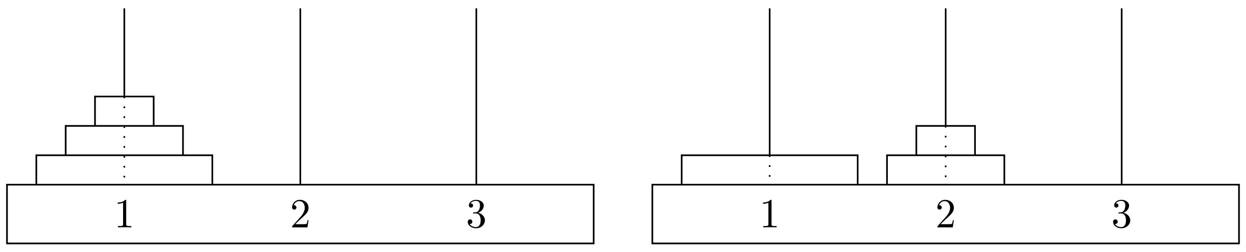

Figure 1: Hanoi Towers in different positions.

In Figure 1, left position is the starting position for $n = 3$ disks. Right position can **not** be reached from left position in **one** move, as you can only move disks one by one. It can however be reached in three moves:

$$1 \to 3, 1 \to 2, 3 \to 2.$$

---

1. Solve the problem for 4 and then for 5 disks.
2. Do you think it is possible to solve the problem for any number n of disks?
3. *(Optional)* Prove or find a counter-example to the following statement:

For all $n \in \mathbb{N}$, the problem can be solved in exactly $2^n - 1$ moves.

Solution

1. Solution for 5 disks:
   1. Move 4 disks from the left rod to the middle one:

      $$1 \to 3; 1 \to 2; 3 \to 2; 1 \to 3; 2 \to 1; 2 \to 3; 1 \to 3; 1 \to 2;$$
      $$3 \to 2; 3 \to 1; 2 \to 1; 3 \to 2; 1 \to 3; 1 \to 2; 3 \to 2$$
   2. Move big disk from the left rod to the right one:

      $$1 \to 3$$
   3. Move 4 disks from the middle rod to the right one:

      $$2 \to 1; 2 \to 3; 1 \to 3; 2 \to 1; 3 \to 2; 3 \to 1; 2 \to 1; 2 \to 3;$$
      $$1 \to 3; 1 \to 2; 3 \to 2; 1 \to 3; 2 \to 1; 2 \to 3; 1 \to 3$$
2. Yes it is, with for example a recursive algorihm. We call `Hanoi(n, x, y)` an algorithm that moves the top n disks from rod x to rod y. It only works if those top n disks are the smallest disks on all rods.

```
Hanoi(n, x, y):
  if n = 1:
    x → y
  else:
    let z = third_rod
    Hanoi(n-1, x, z)
    x → y
    Hanoi(n-1, z, y)
```

3. We prove by an induction on $n \in \mathbb{N}$ the slightly more powerful property:

   For all $n \in \mathbb{N}$, if the n smaller disks on the rods are on the same rod x, it is always possible to move them from x to another rod y in $2^n - 1$ moves.

   Base case, $n = 1$ It is, with one single move $x \to y$.

   Induction case Assume the property is true for $n \in \mathbb{N}$. Let z the rod different from x and y. Assume the $n + 1$ smallest disks are all on the same rod x. Do the following:
   1. Move the top n disks from x to z in $2^n - 1$ moves. It is possible by induction hypothesis.
   2. Move the $(n + 1)^{\text{th}}$ smallest disk from x to y.
   3. Move the top n disks from z to y in $2^n - 1$ moves. It is possible by induction hypothesis.

   This makes $2^n - 1 + 1 + 2^n - 1 = 2^{n+1} - 1$ moves in total. ■

   "Solving the problem" can be achieved by applying this property to the starting position, with $x = 1$ and $y = 3$.

1. What is recursion?

Recursion is a key notion in computer science. It allows elegant definitions and efficient problem solving.

Definition: Recursion

During a procedure, **recursion** happens when one step plans to execute the procedure itself.

Why talking about recursion? Recursive definitions is one key technique among others in defining several formal language tools. Among them, *grammars*, used to described the code structure of programming languages.

Although recursion is a major brick of computer science, you might have already met it in everyday life.

- Some yoghurts or other fermented food are cooked using the result of the recipe as an ingredient.
- Two mirrors facing each other can create an “infinity mirror” phenomenon. If mirror 1 faces mirror 2, then the full reflection of mirror 1 includes the full reflection of mirror 2, which itself includes the full reflection of mirror 1 again.
- Fractals are nice-looking recursive geometric shapes, that are self-similar when you zoom-in, and can occur in nature.

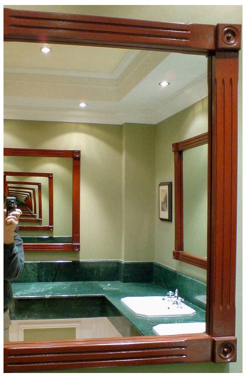

Figure 2: Infinity mirror.

CC-BY 2.0, picture from Brian Snelson on Wikimedia Commons.

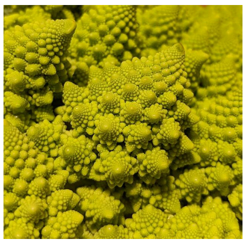

Figure 3: Romanesco broccoli.

CC-BY 4.0, part of a picture from Mds08011 on Wikimedia Commons.

In most cases, recursion is too tedious to be used in everyday life by humans, so it is not that much. Computers, however, are basically designed to only do a gigantic amount of silly and tedious operations, so recursion perfectly fits.

1.1. Recursion as problem solving method

In computer science, it often happens, for a problem $\mathcal{P}$ involving large quantities or large numbers N, that the problem is actually quite easy to solve if you assume you can already solve it for number or quantities just a bit less than N. Here we describe a few such problems.

Exercise: Towers of Hanoi

Assume you can move the top n disks of a stack from one rod to another. How easy is it to move the $n + 1$ top disks from one rod to another?

Solution

See on the solution of the activity.

Exercise: Sorting a list

Assume you want to sort a list of n integer in increasing order, and you know how to sort a list of $n - 1$ integers. How would you proceed using this knowledge?

Same question if you rather know how to sort a list of $\frac{n}{2}$ integers.

Solution

1. Recursively sort the $n - 1$ elements of the list, then read it until you find where to place the n^{th} element. This is called "insertion sort", and is typically used when playing card games.
2. Recursively sort the two halves of the list. When done, you can easily find the smallest element of each list. Build a third list that is the merge of the two by repeatedly picking the smallest element of the two list heads. This is the "merge sort", an efficient sort that is often implemented in standard libraries of programming languages.

Exercise: Change-making problem

The change-making problem consists in, given some $n \in \mathbb{N}$, giving n € with as few coins and bills as possible.

Assume that, for some $N \in \mathbb{N}$, you know how to give the minimum amount of coins and bills to give back n € for any $n \leq N$. How would you proceed to give $N + 1$ €?

Solution

Let $n_a, n_b, \ldots, n_z$ the values of each coin and bill you can return. Compute the amount of coins and bills you must give to return $N - n_a, N - n_b, \ldots, N - n_z$. If $N - n_x$ is the one that can be returned with the lowest amount of coins and bill, give the coin or bill that amounts n_x, then apply the algorithm recursively to the value $N - n_x$.

1.2. Recursion as a definition method

Recursion can also be used as a very elegant definition method. The pattern of such definitions is the following:

1. One or more non-recursively defined objects. They are called **base cases**.
2. Recursive construction rules to describe more complex objects.

The aim of base cases is to be elementary bricks, on top of which are generated more complex objects with recursive rules.

Such definitions do **not** allow infinite objects, or infinite "unfoldings". Each object "includes" at some point of its definition one base case, and often more than one.

One (mathematical) example of an elegantly recursively designed function is the factorial function:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n * (n-1)! & \text{if not.} \end{cases}$$

Another concept that is recursively defined and closer to computer science is trees. Most computer file managers are structured according to a tree structure. The grammar behind most languages spoken by humans can be described by trees. All programming language also interpret or compile code as a tree.

Definition: Tree

A **tree** is either:

- A leaf L, or
- A node $N(t_1, ..., t_n)$, where $t_1, ..., t_n$ are also trees. They are called **children**.

The first node of a tree is called its **root**.

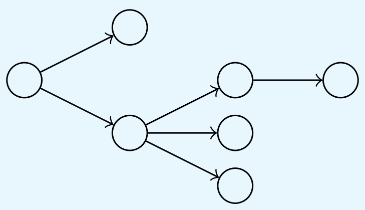

Figure 4: The tree $N(L, N(N(L), L, L))$.

Remarks

- Depending on the needs, leafs and nodes can be labelled, with for example strings or integers. In this case, they are noted $L(\text{label})$ or $N(\text{label}, t_1, ..., t_n)$.
- Sometimes the order between $t_1, ..., t_n$ is important, and sometimes it is not, depending on the context.
- As the inline writing is tedious and not easily human-readable, it is not used often.
- Trees are often drawn by computer scientists with their leaves at the bottom.

Example: The tree behind the code

```
<html>
  <head>
   <title>My super website</title>
  </head>
  <body>
    <h1>I love pastries</h1>
    <h2>and sleeping!</h2>
  </body>
</html>
```

Listing 1: Some HTML code.

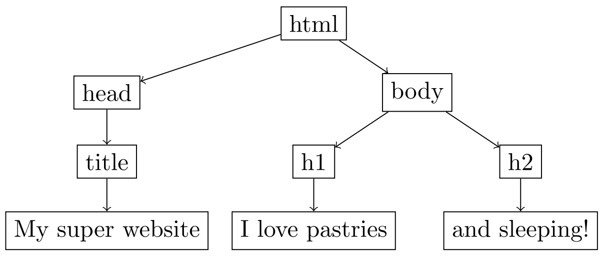

Figure 5: The tree structure behind the code in Listing 1.

In this example, the order between nodes of the tree is important: the `head` part of the code occurs before the `body` part.

Base cases are important In the tree definition, removing the base case (the leaf) makes impossible to generate finite trees. In the factorial function, removing the base case ($0!$) makes the function compute endlessly any value.

2. Some formal language vocabulary

The few following definitions are the basis on top of which is built all the formal languages field.

Definition: Alphabet, word, language

An **alphabet** is a finite set, often denoted by the letter Σ. Its elements are called **letters**, often denoted by… letters.

A **word** is a finite sequence of letters. The **empty word**, denoted ε, is the word with 0 letter. A **language** is a (non necessarily finite) set of words.

Example

- $\Sigma_1 = \{a, b, ..., z\}$ is an alphabet, *computer* is a word on Σ_1, and the set of English words is a language on Σ_1.
- $\Sigma_2 = \{0, 1\}$ is an alphabet, and any data stored on a computer can be ultimately expressed as a word on Σ_2.
- $\Sigma_3 = \{A, T, C, G\}$ is an alphabet, and your genome is a word on Σ_3.

Pay attention to the fact that a word or a language is **always** defined **on an alphabet**. Alphabet can be omitted when mentioning a word or a language if it is obvious.

Definition: Concatenation

Concatenation is a very often used operation on formal languages.

Let two words w_1 and w_2 on a same alphabet Σ. The **concatenation between two words** w_1 and w_2, often denoted $w_1 \cdot w_2$ or $w_1 w_2$, is a new word which starts with the letters of w_1 in order, followed by the letters of w_2 in order.

A concatenation of n occurrences of a word w can be denoted w^n.

Let two languages L_1 and L_2. The **concatenation between two languages** L_1 and L_2, often denoted $L_1 \cdot L_2$ or $L_1 L_2$, is the set

$$\{w_1 \cdot w_2 \mid w_1 \in L_1, w_2 \in L_2\}.$$

A concatenation of n occurrences of a language L can be denoted L^n.

Examples
- The concatenation between *super* and *computer* is the word *supercomputer*.
- If $w_1 = TATA$ and $w_2 = GAGA$, then $w_1 w_2 = TATAGAGA$.
- If $L_a = \{a, aa\}$ and $L_b = \{b, bb\}$, then $L_a \cdot L_b = \{ab, abb, aab, aabb\}$.

3. Regular expressions, Regular languages

Regular expressions is a syntactic tool to describe *regular languages*, a central class of formal languages.

What does *syntactic* mean? In computer science, when describing new objects (e.g. new programming languages), there is often a difference between *syntax* and *semantic*.
- ***Syntax*** describes the ***structure*** of the object. e.g. for programming languages: what are the keywords, when to use characters like `()`, `{}` or `[]`, is spacing allowed. In other words, which source codes should be recognized by the machine as a program in this language, and which should not.
- ***Semantic*** describes the ***meaning*** of the object. e.g. for programming languages: how are implemented integers, additions, recursive calls, interaction with memory. Is spacing significant. What should happen when writing `1 + "1"`. Is logical or lazy or not.

Why talking about regular expressions and regular languages? Later in this course, you will learn some quantum programming languages syntax and semantic. Regular expressions and languages is a first step in understanding how deeply tied are syntax and semantic. It also happens to be closely related to finite automata, a simple conceptual computing machine presented in next lecture.

3.1. Regular languages

A major question in formal languages is: how to *define* rich or precise languages (e.g. natural languages, programming languages, technical patterns as email addresses) precisely enough so that it matches the reality, but simply enough so that it can be easily manipulated and understood, either by humans or computers. *Regular languages* is a class of languages that ticks many of theses boxes.

One possible first definition of “regular languages” is the following. Let Σ an alphabet, and $\mathcal{C}(\Sigma)$ the class of wannabe regular languages on Σ. $\mathcal{C}(\Sigma)$ is the smallest set such that:

- The empty set $\emptyset$ is in $\mathcal{C}(\Sigma)$.
- The language consisting in only the empty word $\{\varepsilon\}$ is in $\mathcal{C}(\Sigma)$.
- For each $l \in \Sigma$, the language $\{l\}$ consisting in the one-letter word l only is in $\mathcal{C}(\Sigma)$.

Those are the base cases of the definition. Next we need to allow some richer pattern. This is the aim of the following rules:

- If L_1 and L_2 are in $\mathcal{C}(\Sigma)$, then their union $L_1 \cup L_2$ also is.
- If L_1 and L_2 are in $\mathcal{C}(\Sigma)$, then their concatenation $L_1 \cdot L_2$ also is.

This definition allows defining precise languages… but still, there is some nice possibility missing. Can you guess what?

Exercise: Regular languages?

1. Let L in $\mathcal{C}(\Sigma)$. Show that L is a finite language.
2. Let L a finite language. Show that L is in $\mathcal{C}(\Sigma)$.

Solution

1. The following proof is said *by structural induction.*
    1. If L is a base case of the $\mathcal{C}(\Sigma)$ construction, it is indeed finite.
    2. Assume L_1 and L_2 are finite languages in $\mathcal{C}(\Sigma)$, with n_1 and n_2 elements respectively. Then $L_1 \cup L_2$ has at most $n_1 + n_2$ elements, and
    $L_1 \cdot L_2$ has at most $n_1 \cdot n_2$. They are both finite.

    This concludes the proof.
2. By induction on the number of words n in the finite language L.
    Base case, $n = 0$ $\emptyset$ is in $\mathcal{C}(\Sigma)$.
    Induction case Let L a finite language with $n > 0$ words. Consider a word $w \in L$. The language $L \setminus \{w\}$ has $n - 1$ words, so by induction hypothesis it is in $\mathcal{C}(\Sigma)$.

    The language $\{w\}$ also is in $\mathcal{C}(\Sigma)$, because if w writes $w_1 w_2 ... w_k$ then $\{w\}$ is

    $$\{w_1\} \cdot \{w_2\} \cdot ... \cdot \{w_k\}$$

    Thus $L = (L \setminus \{w\}) \cup \{w\}$ is in $\mathcal{C}(\Sigma)$.

One key property missing in $\mathcal{C}(\Sigma)$ is that it can not contain infinite languages. As alphabets Σ are always finite, this induces that a language in $\mathcal{C}(\Sigma)$ can not contain words of arbitrarily large size *(prove it!)*. One elegant way to lift this constraint, without adding too much complexity, is to allow repeated concatenations.

Definition: Kleene Star

Let L a language. The **Kleene Star** of L, noted L^*, is the set:

$$\bigcup_{n \in \mathbb{N}} L^n$$

where $L^0 = \{\varepsilon\}$ by convention.

In other words, L^* contains any finite, arbitrarily large sequence of concatenations of words of L. It does **not** contain any infinite word.

This final piece of the puzzle allows to entirely define the class of regular languages:

Definition: Regular languages

Let Σ an alphabet. The class of **regular languages on Σ**, denoted $\mathrm{Reg}(\Sigma)$, is the smallest set such that:

- The empty set $\emptyset$ is in $\mathrm{Reg}(\Sigma)$.
- The language consisting in only the empty word $\{\varepsilon\}$ is in $\mathrm{Reg}(\Sigma)$.
- For each letter $l \in \Sigma$, the language $\{l\}$ consisting in the one-letter word l only is in $\mathrm{Reg}(\Sigma)$.
- If L_1 and L_2 are in $\mathrm{Reg}(\Sigma)$, then their union $L_1 \cup L_2$ also is.
- If L_1 and L_2 are in $\mathrm{Reg}(\Sigma)$, then their concatenation $L_1 \cdot L_2$ also is.
- If L is in $\mathrm{Reg}(\Sigma)$, then L^* also is.

Regular languages can be described with natural languages, e.g. "The set of words starting with an a and ending with a b.", but this is not always convenient. One easier way to describe complex regular languages is *regular expressions*.

3.2. Regular expressions

Regular expressions is a convenient *syntactic* tool to describe regular languages, both in a human-readable and computer-readable way.

Definition: Regular expression

A *regular expression* is a tree labelled with characters. Let Σ an alphabet. The set of **regular expressions on Σ**, denoted $\mathrm{Regex}(\Sigma)$, is the following:

- The tree $L(\emptyset)$ is in $\mathrm{Regex}(\Sigma)$.
- The tree $L(\varepsilon)$ is in $\mathrm{Regex}(\Sigma)$.
- For each letter $l \in \Sigma$, the tree $L(l)$ is in $\mathrm{Regex}(\Sigma)$.
- If e_1 and e_2 are in $\mathrm{Regex}(\Sigma)$, then the tree $N(\cup, e_1, e_2)$ also is.
- If e_1 and e_2 are in $\mathrm{Regex}(\Sigma)$, then the tree $N(\cdot, e_1, e_2)$ also is.
- If e is in $\mathrm{Regex}(\Sigma)$, then the tree $N(*, e)$ also is.

Remarks

- Regular expressions are almost always denoted as sequences of characters rather than trees. See example in Figure 6.
- Parenthesis can be added to remove ambiguity, they are silent characters.
- Regular expressions are only trees labelled with characters, and do not have any meaning by themselves.

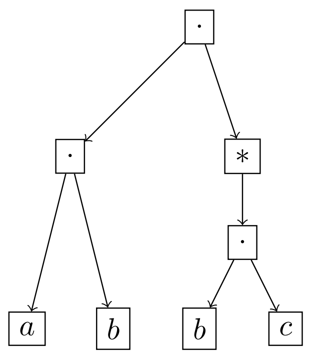

Figure 6: The sequence of characters $ab(bc)^*$ denotes this expression.

- When written inline, a Kleen star with no parenthesis only matches the previous character:

$$aa^* = a(a)^* \neq (aa)^*$$

The meaning given to regular expressions – in other word, their *semantic* – is the following:

Definition: Language of a regular expression

Let e a regular expression on Σ. The **language of e**, denoted $\mathcal{L}(e)$, is the following:
- If $e = L(\emptyset)$, then $\mathcal{L}(e) = \emptyset$.
- If $e = L(\varepsilon)$, then $\mathcal{L}(e) = \{\varepsilon\}$.
- If $e = L(l)$, then $\mathcal{L}(e) = \{l\}$.
- If $e = N(\cup, e_1, e_2)$, then $\mathcal{L}(e) = \mathcal{L}(e_1) \cup \mathcal{L}(e_2)$.
- If $e = N(\cdot, e_1, e_2)$, then $\mathcal{L}(e) = \mathcal{L}(e_1) \cdot \mathcal{L}(e_2)$.
- If $e = N(*, e')$, then $\mathcal{L}(e) = \mathcal{L}(e')^*$.

Remarks
- If this semantic seems obvious to you, it is because the *syntax* of regular expressions has been chosen to match the symbols used by the mathematical operators. This is rather good news: a confusing syntax is source of errors!
- Different regular expressions can have the same semantic : $\mathcal{L}(aa^*) = \mathcal{L}(a^*a)$.
- Concerning regular expressions, the tree structure is often not that interesting, whereas the semantic of the tree is. As such, when using inline regular expressions, equality denotes the equality between languages rather than the equality of the trees. According to this abuse of notation, $aa^* = a^*a$, even though the trees are different. This abuse of notation is very local, and is forbidden when manipulating other objects like grammars.

Exercise: Regular expressions semantic is regular languages

1. Let e a regular expression on Σ. Prove that $\mathcal{L}(e)$ is a regular language on Σ.
2. Let L a regular language on Σ. Prove that there exist a regular expression $e \in \text{Regex}(\Sigma)$ such that $\mathcal{L}(e) = L$. Is it unique?

Solution

1. By induction on the regular expression structure.
    1. If e is a base case of the regular expression construction, its language is regular.
    2. Assume e_1 and e_2 are regular expressions whose languages, $\mathcal{L}(e_1)$ and $\mathcal{L}(e_2)$, are regular. Then $\mathcal{L}(e_1 \cup e_2) = \mathcal{L}(e_1) \cup \mathcal{L}(e_2)$ also is. The proof is similar for $e_1 \cdot e_2$ and e_1^*.
2. By induction on the regular language structure.
    1. If L is a base case of a regular language, there exist a (base case of a) regular expression whose language describes L.
    2. Assume L_1 and L_2 are regular languages described by regular expressions e_1 and e_2. Then $L_1 \cup L_2$ is described by $e_1 \cup e_2$. The proof is similar for $L_1 \cdot L_2$ and L_1^*.

The expression is not unique: if e has L as a language, then $e \cup \emptyset$ also does.

Regular expressions make describing regular languages a lot more convenient.

Exercise: Regular expressions

1. Let Σ an alphabet. Describe the set of all words on Σ with a regular expression.
2. Let $\Sigma_{\text{int}} = \{0, 1, 2, ..., 9\}$. Describe with a regular expression the language of decimal notations of integers divided by 5.
3. Find a relevant alphabet to describe the format of valid email addresses ending with `.org`, and write the regular expression that describes them.
4. Let $\Sigma = \{(,)\}$. Can you find a way to describe the set of balanced strings of brackets with a regular expression? If not, can you explain why?

Solution

1. Let $a, b, \ldots, z$ the letters in Σ. $(\{a\} \cup \{b\} \cup \ldots \cup \{z\})^*$ fits. It is often written Σ^*.
2. Integers divided by 5 end with 0 or 5, so $(\Sigma_{\text{int}})^* \cdot (\{0\} \cup \{5\})$ fits.
3. Consider alphabet $\Sigma_{\text{ln}} = \{\text{a, b, …, z, 0, 1, …, 9}\}$ the alphabet of letters and numbers. Consider the alphabet $\Sigma = \Sigma_{\text{ln}} \cup \{., @\}$. Then one possible pattern is the following:

$$((\Sigma_{\text{ln}})^*.)^*\Sigma_{\text{ln}}@((\Sigma_{\text{ln}})^*.)^*\Sigma_{\text{ln}}\{.\text{org}\}$$

4. This is impossible. The informal explanation is that when checking if a world is in the language of a regular expression, you must read the expression from left to right, and the only memory you can have lies in the finitely many characters of the expression. So you can not count arbitrarily high how many parenthesis you have met, and no matter how complex your expression is, there will be a point when it either will contain the word

$$(^n)^n$$

or not contain the word

$$(^n)^{n-1}.$$

Do not panick if you did not get this explanation. It will become more clear when you will know finite automata, and how tied with regular expressions they are.

4. Grammars

Why talking about grammars? In regular expressions, $\cup$ and $\cdot$ are associative operators:

$$(a \cup b) \cup c = a \cup (b \cup c)$$

so the syntactical detailed structure of a regular expression does not matter much. There are however some application cases where the structure of what you are analyzing does matter – e.g. analyzing source code. For those cases, there exist a more powerful tool: grammars.

Informal description Grammars is a tool designed to describe hierarchical organization between “blocks” of things with different properties.

Why this name? The name of this tool comes from linguistic, where it is used to describe what sentences are syntactically correct.

Natural language example Consider the following sentence:

I am glad that I am here, and so are you.

It can have two very different meanings:

1. I am glad that I am here. You are glad that you are here.
2. I am glad that both me and you are here.

Distinguishing between the two depends on how the sentence is built. Consider that you have the following *construction rules* to build a syntactically correct sentence:

$$\begin{aligned} &S \to NVS && N \to I \mid \text{you} \\ &S \to NV && V \to \text{am glad that} \mid \text{am here} \mid \text{are} \\ &S \to S \text{ and so } VN && \end{aligned}$$

Bars read as an "or", and are a shorthand to write several rules in one. To build a sentence, start with the S symbol, and then replace symbols on the left of a production rule arrows by things on the right. Example:

$$S \to NV \to N \text{ am here} \to \text{I am here}$$

With those rules, you can create the example sentence with two different *syntax trees* depicted in Figure 7. The one on the left corresponds to the first interpretation, the one on the right corresponds to the second one.

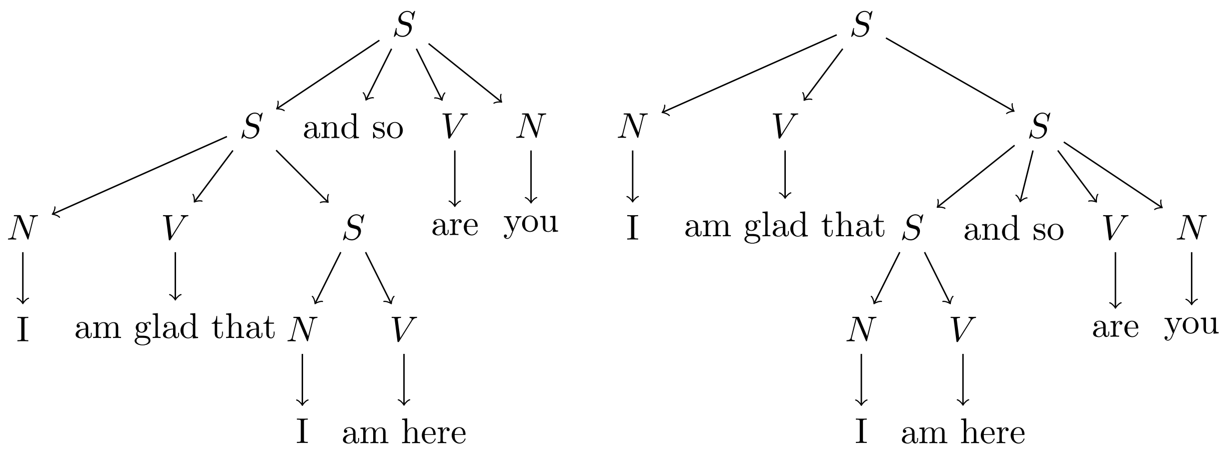

Figure 7: Two meanings of "I am glad that I am here, and so are you."

This kind of ambiguity is often not much of a problem in natural language, as either the context gives the explanation, or you can ask the person you are speaking to. In programming languages, however, you should absolutely avoid letting program chose between two very different meanings of your code. This would result in very unconsistent and untrustable programs.

What use for grammars in programming languages? Each modern programming language includes a grammar in its design. When processing source code, they start by generating a tree that describes the code according to their grammar.

Definition: Grammar

A **grammar** is a tuple $G = (V, \Sigma, P, S)$ where:
- V is an alphabet of so-called *non-terminal* symbols, or *variables.*
- Σ is an alphabet of so-called *terminal* symbols. It has no common symbol with V.
- $P \subseteq (V \cup \Sigma)^* V (V \cup \Sigma)^* \times (V \cup \Sigma)^*$ is a *finite* set of so-called *production rules.*
- $S \in V$ is the *starting symbol.*

When each production rule of a grammar is in $V \times (V \cup \Sigma)^*$, the grammar is called *context-free.*

Definition: Derivation

Let $G = (V, \Sigma, P, S)$ a grammar. A **derivation** is a tree generated by a sequence production rules of G from its starting symbol. We say that a word $w \in \Sigma \cup V$ can be *derived* from a word or symbol u, if there exist a derivation starting from u. If no symbol u is specified, consider the starting symbol of G.

The formal definition is a bit heavy, and grammars are often described by the list of their production rules.

Example grammar The following example is a context-free grammar describing a language designed to achieve computations without loops:

$E \to E + E$	$B \to E = E$	$P \to P; P$
$E \to E * E$	$B \to E < E$	$P \to \texttt{if } B \texttt{ then } P \texttt{ else } P$
$E \to N$	$B \to \neg B$	$P \to V := E$
$E \to V$	$B \to B \lor B$	$P \to \texttt{return } E$
$N \to NN \mid 0 \mid 1 \mid \dots \mid 9 \mid \varepsilon$		$V \to x \mid y \mid z$

Listing 2: A toy programming language grammar[1].
It can generate **E**xpressions, **V**ariables, **N**atural numbers, **B**ooleans, and **P**rograms.

Here the starting symbol to create a program is always P. Although the symbols used here should look familiar to you, keep in mind that a grammar only gives the *structure* of the code, and not is meaning. Without semantic, it might be that $+$ means concatenation and $*$ describes tuples… Defining semantic becomes even more important when designing quantum programming languages, where symbols like $\otimes$, $⅋$ or $\multimap$ might be used, and if so should have their meaning clearly defined.

[1]Syntactical ambiguity here!

Exercise: Programming language grammar

For this exercise, consider production rules described in Listing 2. For each of the following strings, count the number of derivation that can produce them from P. **Pay a close attention** to what are the *exact* production rules you are allowed to use!

1. $x + y * z$
2. `if` 1 `then` $x := 3 * x + 1$ `else` $x := x/2$
3. `return` x; `return` y
4. $x := 0$

When there are not exactly one, can you think of a way to make sure that exactly one derivation is possible?

Solution

1. Two possible trees: one with $+$ at the root, and one with $*$. Adding parenthesis to the expression or having priority rules can reduce this number to one.
2. Impossible, for several reasons:
   - No `/` symbol in the production rules;
   - No `1` allowed as boolean.

   Adding new rules can solve this.
3. Only one possible tree.
4. Infinitely many possible trees, because you can create arbitrarily many N characters with the $N \to NN$ rule, and make them vanish by using the $N \to \varepsilon$ rule. One possible way to fix this is to change the N rules for:

$$N \to 0N \mid 1N \mid \ldots \mid 9N \mid \varepsilon.$$