Thursday - August 8, 2024

Introduction to Finite State Automata

Exercise: recursive expressions

Recall in the last lecture, we define the type `Numb` to be the following:

```
data Numb = Z | S Numb deriving Show
```

- Write a function `lessThanTwo :: Numb -> Bool` which returns `True` if a number is less than two (i.e., `S Z` or `Z`), and `False` otherwise.

```
lessThanTwo :: Numb → Bool
lessThanTwo = \n → case n of
                     Z → True
                     S n' → case n' of { Z → True ; S n'' → False }
```

peel of one S — n is — we have more than one S so > 2

- Write a function `add :: Numb -> (Numb -> Numb)` which computes the sum of two numbers.

```
add :: Numb → Numb → Numb
add :: \n → \m → case n of
```

ex: add (S Z) (S Z) — n is 0 — Z → m — n is greater than 1 — S n' → S (add n' m) — until we hit zero

S n' → S (add Z m) — S

- `map`: This function takes as inputs another function f and a list and returns another list containing the output of the function applied to every element in the list. This is a built-in function in `Prelude`. To practice, let's write our own implementation of the `map` function.
  - `map (\x ->x^2) [1,2,3,4,5,6] => [1,4,9,16,25,36]`
  - `map (\x ->x-1) [2,3,4] => [1,2,3]`

```
map2 :: (a → b) → [a] → [a]
map2 = \f → \l → case l of
                   [] → []
                   x:: rest → (f x) : (map2 f rest)
```

could do apple:: bananas —

1 Formal Language theory

Formal language theory (FLT) is concerned with the mathematical properties and computational complexity of formal languages and the formal grammars that describe them [1]

1.1 Basic notation

- An **alphabet** Σ is a set of discrete symbols, typically called letters.
- A **string** is a sequence of letters from an alphabet. The empty string ϵ is the string with no symbols (so its length is 0; or $|\epsilon| = 0$)
- We can characterize a **language** $\mathcal{L}$ as a set of strings formed over an alphabet Σ.
- A language can be described, or generated, by a **grammar** G, which consists of rules for forming strings from the language's alphabet.
- A *class of languages* is a set of languages.

Examples of formal languages:

(1) a. $\Sigma_1 = \{a, b\}$
$\mathcal{L}_1 = \{a, b, ab, ba\}$

b. $\Sigma_2 = \{0, 1\}$
$\mathcal{L}_2 = \{01, 0011, 000111, ...\}$

c. $\Sigma_3 = \{\text{hot, dog}\}$
$\mathcal{L}_3 = \{\text{hot, dog, hotdog}\}$

1.2 Automata

There is a tight connection between Formal Language Theory and Automata Theory.

- **Automata** (singular: automaton) are abstract machines that perform computations on an input by transitioning through a series of states.
- Certain types of automata can recognize/generate strings of certain kinds of languages, characterizable in terms of the grammars required to describe them.
- Thus, there is a tight connection between formal grammars and automata, which we can exploit for various purposes.

[1] Basic knowledge of set theory (sets, powersets, subset relations, etc) is assumed. Please read the Appendix for a refresher. If you have never taken any classes that cover basic concepts in set theory, feel free to talk to me, and I can go over the appendix with you.

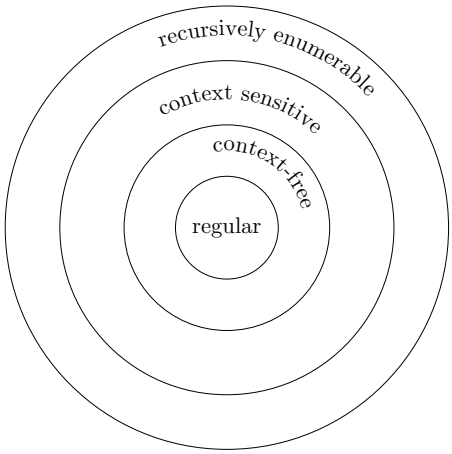
Chomsky (1956)[2] described a hierarchy of formal grammars and languages:

(2)

Grammar	Language	Automaton
Type-0	Unrestricted/recursively enumerable	Turing machine
Type-1	Context-sensitive	Linear-bounded automaton
Type-2	Context-free	Nondeterministic pushdown automaton
Type-3	Regular	Finite-state automaton

less powerful ↓

(3)



← subsets

Chomsky Hierarchy: regular ⊆ context-free ⊆ context-sensitive ⊆ unrestricted

Where do natural languages sit?

Do the grammars for combining sounds (phonology), combining words (morphology), forming sentences (syntax), and computing meanings (semantics) require the same amount of computational power?

Remarks

- We'll start with regular languages and finite-state automata, and over the next few weeks, we will gradually work our way up the Chomsky Hierarchy.
- FLT provides a way of concretely characterizing computational problems (in our case, aspects of natural language) in a way that allows us to explore their mathematical regularities.

  ↪ Everything that you have learned in (formal) linguistics courses can be cast as formal grammars, even though notation and such will differ in practice.

  ↪ e.g., the transformational generative grammar (Chomsky, 1965)[3] is Turing-complete (Peters & Ritchie 1973)[4]

[2]Chomsky, N. (1956). Three models for the description of language. *IRE Transactions in Information Theory*, 2:113–124

[3]Chomsky, N. (1965). *Aspects of the theory of syntax*. MIT Press, Cambridge, MA

[4]Peters Jr, P.S. and Ritchie, R.W., 1973. On the generative power of transformational grammars. Information sciences, 6, pp.49-83.

2 Finite-state automata

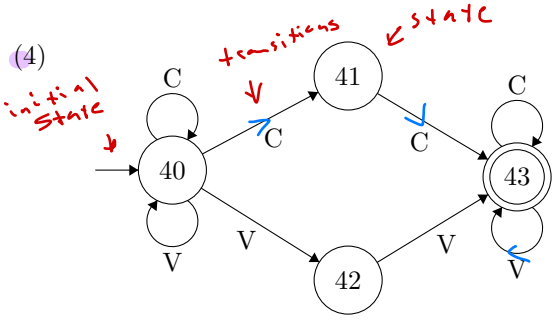
2.1 Informal definition

Below are graphical representations of several finite-state automata (FSAs).

- The circles represent **states**.
- The **initial state** is indicated with an "arrow from nowhere."
- **Ending states** (also called **final** or **accepting** states) are indicated by a double circle.
- The arrows represent **transitions** from one state to another based on a particular input.

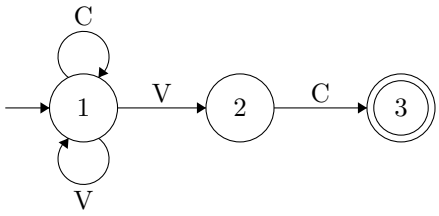
The FSA in (4) generates the strings from the alphabet $\{C, V\}$ consisting of all strings (and only those strings) that contain either two adjacent 'C's or two adjacent 'V's (or both).

(4)



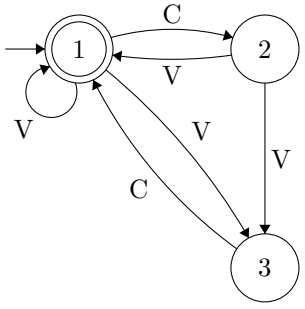
The FSA in (5) generates strings which end in 'VC.'

(5)



If we think of state 1 as indicating syllable boundaries, then the FSA in (6) generates sequences of syllables of the form '(C)V(C)'. The string 'VCV', for example, can be generated via two different paths, 1-1-2-1 and 1-3-1-1, corresponding to different syllabifications.

(6)



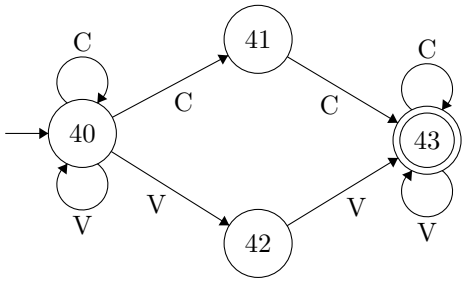
2.2 Formal definition of an FSA

A finite-state automaton (FSA) is a five-tuple $(Q, \Sigma, I, F, \Delta)$ where:

1. Q is a **finite** set of states;
2. Σ, the alphabet, is a finite set of symbols;
3. $I \subseteq Q$ is the set of initial states;
4. $F \subseteq Q$ is the set of ending states;
5. $\Delta \subseteq Q \times \Sigma \times Q$ is the set of transitions.

So, strictly speaking, (4) is a graphical representation of the following mathematical object:

(7) $\langle \{40, 41, 42, 43\}$, finite set of states
$\{\text{C}, \text{V}\}$, alphabet
$\{40\}$, initial
$\{43\}$, final
$\{(40, \text{C}, 40), (40, \text{C}, 41), (40, \text{V}, 40), (40, \text{V}, 42),$
$(41, \text{C}, 43), (42, \text{V}, 43),$
$(43, \text{C}, 43), (43, \text{V}, 43)\}\rangle$
transitions (think of as paths)



2.3 Recognizing or accepting a string

For M to generate a string of three symbols, say $x_1x_2x_3$, there must be four states q_0, q_1, q_2 and q_3 such that:

- $q_0 \in I$, and (initial)
- $(q_0, x_1, q_1) \in \Delta$, and
- $(q_1, x_2, q_2) \in \Delta$, and (transitions)
- $(q_2, x_3, q_3) \in \Delta$, and
- $q_3 \in F$. (final)

same thing

These requirements generalize as such:

M generates a string of n symbols, say $x_1x_2...x_n$, iff there are $n+1$ states $q_0, q_1, q_2...q_n$ such that:

1. $q_0 \in I$, and
2. for every $i \in \{1, 2, ...n\}$, there is a transition $(q_{i-1}, x_i, q_i) \in \Delta$, and
3. $q_n \in F$.

Now, let us look at the automaton above. Does this automaton generate the string 'VCCVC'?

40 V 40 C 41 C 43 V 43 C 43 ✓

2.4 Defining generation recursively

We will write $\mathcal{L}(M)$ to represent the set of strings generated by an FSA M. Thus, a rough definition for membership in $\mathcal{L}(M)$ is as follows:

(8) $w \in \mathcal{L}(M)$

$$\Leftrightarrow \bigvee_{\text{all possible paths } p} \Big[\text{string } w \text{ can be generated by path } p\Big]$$

$$\Leftrightarrow \bigvee_{\text{all possible paths } p} \Bigg[\bigwedge_{\text{all steps } s \text{ in } p} \Big[\text{step } s \text{ is allowed and generates the appropriate part of } w\Big] \Bigg]$$

It's handy to write $I(q_0)$ in place of $q_0 \in I$, and likewise for F and Δ. Then one way to make (8) more precise is:

(9) $x_1 x_2 ... x_n \in \mathcal{L}(M)$

$$\Leftrightarrow \bigvee_{q_0 \in Q} \bigvee_{q_1 \in Q} ... \bigvee_{q_{n-1} \in Q} \bigvee_{q_n \in Q} \Big[I(q_0) \wedge \Delta(q_0, x_1, q_1) \wedge ... \wedge \Delta(q_{n-1}, x_n, q_n) \wedge F(q_n)\Big]$$

But it's practical and enlightening to break this down in a couple of different ways.

2.4.1 Forward values

For any FSA M there's a two-place predicate fwd_M relating states to strings in an important way:

(10) $fwd_M(w)(q)$ is true iff there's a path through M from some initial state to the state q, emitting the string w

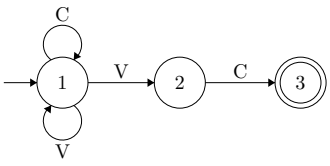
Given a way to work out $fwd_M(w)(q)$ for any string and any state, we can easily use this to check for membership in $\mathcal{L}(M)$:

(11) $w \in \mathcal{L}(M) \Leftrightarrow \bigvee_{q_n \in Q} \Big[fwd_M(w)(q_n) \wedge F(q_n)\Big]$

We can represent the predicate fwd_M in a table. Each column shows fwd_M values for the *entire prefix* consisting of the header symbols to its *left*. The first column shows values for the empty string.

Here's the table of fwd_M values for prefixes of the string 'CVCCVVC' for the FSA in (5):

(12)



State	C	V	C	C	V	V	C
1	1	1	1	1	1	1	1
2	0	0	1	0	0	1	0
3	0	0	0	1	0	0	1

Here is the recursive definition of fwd_M:

(13) a. $fwd_M(\epsilon)(q) = I(q)$

b. $fwd_M(x_1...x_n)(q) = \bigvee_{q_{n-1} \in Q} \left[fwd_M(x_1...x_{n-1})(q_{n-1}) \wedge \Delta(q_{n-1}, x_n, q) \right]$

This suggests a natural and efficient algorithm for calculating these values: write out the table, start by filling in the leftmost column, and then fill in the other columns from left to right. This is where the name "forward" comes from.

2.4.2 Backward values

We can also do all the same things flipped around in the other direction.

For any FSA M there's a two-place predicate bwd_M relating states to strings in an important way:

(14) $bwd_M(w)(q)$ is true iff there's a path through M from the state q to some ending state, emitting the string w

Given a way to work out $bwd_M(w)(q)$ for any string and any state, we can easily use this to check for membership in $\mathcal{L}(M)$:

(15) $w \in \mathcal{L}(M) \Leftrightarrow \bigvee_{q_0 \in Q} \left[I(q_0) \wedge bwd_M(w)(q_0) \right]$

We can represent the predicate bwd_M in a table. Each column shows bwd_M values for the *entire suffix* consisting of the header symbols to its *right*. The last column shows values for the empty string. Here's the table of bwd_M values for suffixes of the string 'CVCCVVC' for the FSA in (5):

(16)

State	C	V	C	C	V	V	C	
1	1	1	1	1	1	1	0	0
2	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	0	1

Similarly, bwd_M can also be defined recursively:

(17) a. $bwd_M(\epsilon)(q) = F(q)$

b. $bwd_M(x_1 ... x_n)(q) = \bigvee_{q_1 \in Q} \left[\Delta(q, x_1, q_1) \wedge bwd_M(x_2 ... x_n)(q_1)\right]$ ← recursive case

q_1 is in set delta?

call backward recursively on rest

2.4.3 Forward and backward values together

Now we can say something beautiful:

(18) $uv \in \mathcal{L}(M) \Leftrightarrow \bigvee_{q \in Q} \left[fwd_M(u)(q) \wedge bwd_M(v)(q)\right]$

true true

And in fact (11) and (15) are just special cases of (18), with u or v chosen to be the empty string:

(19) a. $w \in \mathcal{L}(M) \Leftrightarrow \bigvee_{q \in Q} \left[fwd_M(w)(q) \wedge bwd_M(\epsilon)(q)\right] \Leftrightarrow \bigvee_{q \in Q} \left[fwd_M(w)(q) \wedge F(q)\right]$

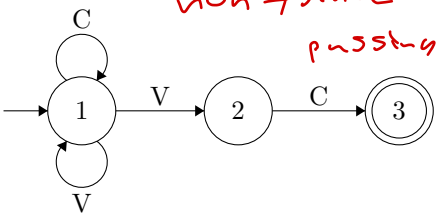
b. $w \in \mathcal{L}(M) \Leftrightarrow \bigvee_{q \in Q} \left[fwd_M(\epsilon)(q) \wedge bwd_M(w)(q)\right] \Leftrightarrow \bigvee_{q \in Q} \left[I(q) \wedge bwd_M(w)(q)\right]$

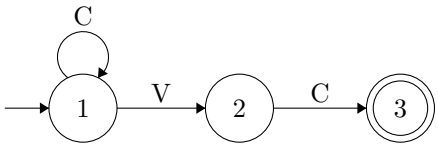
3 Deterministic and non-deterministic FSAs

3.1 Formal definitions

- If each transition in an FSA is uniquely determined by its source state and its input, the FSA is considered **deterministic**.
  ↪ There are never two arcs leading out of the same state that are labeled with the same symbol.
  ↪ Each string corresponds to at most one path through the states.
  ↪ fwd only ever produces singleton sets or the empty set.
- Otherwise, the FSA is considered **non-deterministic**. [5]
  ↪ For any given state and any given symbol, there might be more than one possible next state in a non-deterministic FSA. Or, there might be no possible next state.

For the following two FSAs, which one is deterministic? Which one is non-deterministic?





3.2 Equivalence

You might think that nondeterminism would make an FSA more powerful. However, deterministic FSAs are actually just special cases of nondeterministic FSA.

It turns out that the distinction between deterministic and nondeterministic FSAs is *inconsequential* for generative capacity: deterministic and nondeterministic FSAs generate the same class of languages. [6] This is illustrated by the following procedure for converting any nondeterministic FSA M into an equivalent deterministic FSA M' that generates the same language as M:

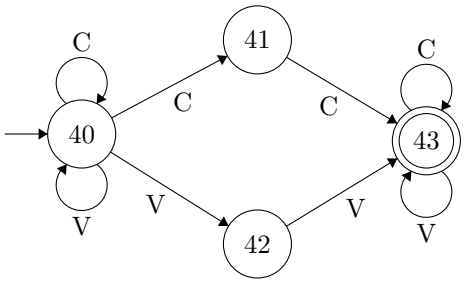
- First, set up the states of M' to correspond to possible **sets of states** in M.
- Then, for any string u, the one state in $fwd'_M(u)$ will be the one corresponding to the set $fwd_M(u)$.
- Working out the transitions of the new automaton M' for a symbol x corresponds to working out how to calculate $fwd_M(ux)$ from $fwd_M(u)$, which we saw today.

[5] This is parallel to the distinction between relations and functions: a relation can have many outputs for a single input, but a function has a single input for a single output. Looking back to our formal definition on p.5, you will notice that Δ is defined as a relation, not as a function! So we are really defining non-deterministic FSAs.

[6] However, this distinction matters for some other variants of finite-state methods, such as finite-state transduction for string-to-string mappings.

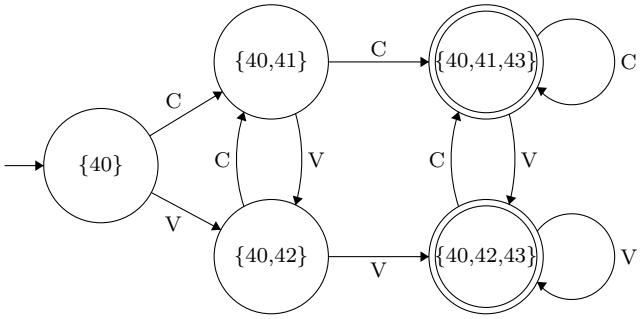
If we apply this determinization procedure to the nondeterministic FSA in (20), we produce the new FSA in (21):

(20)



non

(21)



To see the connection, here's the table of forward values for the string 'CVCCV' using the FSA in (20):

initial

(22)

State	C	V	C	C	V	
40	1	1	1	1	1	1
41	0	1	0	1	1	0
42	0	0	1	0	0	1
43	0	0	0	0	1	1

4 Intersection of FSAs

The stringsets generated by FSAs are **closed under intersection.**
↪ if there is an FSA that generates the stringset L and there is an FSA that generates L', then there is also an FSA that generates the stringset $L \cap L'$.
↪ Intersecting two FSA-describable stringsets will never produce a non-FSA-describable stringset.
↪ Given any two FSAs, we can construct a third FSA that generates the intersection of the original two FSAs' stringsets.

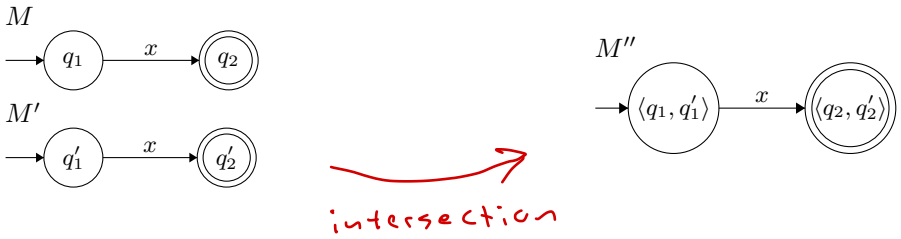
4.1 Informal recipe

The important ideas here will be that:

- the states of the new FSA are pairs, consisting of a state from the first FSA and a state from the second; and
- each transition emitting a symbol x in the new FSA must connect two states which
  1. have as their first components states connected by an x-emitting transition in the first FSA, and
  2. have as their second components states connected by an x-emitting transition in the second FSA.

The new FSA will "simulate" the workings of both of the two original FSAs simultaneously. Here's the picture to have in mind. M'' generates the intersection of the stringsets that are generated by M and M'.

(23)



4.2 Formal recipe

The general "recipe" for intersecting two FSAs is as follows:

(24) Given two FSAs $M = (Q, \Sigma, I, F, \Delta)$ and $M' = (Q', \Sigma, I', F', \Delta')$, we can construct an FSA M'' that will generate $L(M) \cap L(M')$ as follows:

a. $M'' = (Q \times Q', \Sigma, I \times I', F \times F', \Delta'')$
b. $(\langle q_1, q'_1 \rangle, x, \langle q_2, q'_2 \rangle) \in \Delta''$ iff both $(q_1, x, q_2) \in \Delta$ and $(q'_1, x, q'_2) \in \Delta'$ transitions

"transition is in both"

4.3 Why bother?

Assume there is a Martian language that has long-distance vowel harmony: vowels must agree in their front/backness and long-distance consonant disharmony: consonants must disagree in their place of articulation. If you know that the restrictions on vowels and on consonants can be represented by separate FSAs respectively, then, you know that to capture the whole-Martian-language phonotactic restrictions, you can intersect them and use a big FSA. More importantly, FSAs are enough, and you do not need more computational resources. Then, it is safe to conclude that "being FSA-recognizable" is an upper bound for the system Martian use to compute sound sequences.

Appendix: Set theory basics

Sets and set operations A *set* is a collection of distinct objects referred to as *elements*. If an object a is an element of a set A, it is denoted as $a \in A$. The symbol $\notin$ denotes non-membership (e.g., $b \notin \{a, c, d\}$). The number of elements in a set A is called the cardinality of A, written $|A|$. A set A is *finite* if and only if (iff) there exists some natural number k such that A has exactly k many elements, as $|A| = k$. One finite set that is of particular interest is the set with no elements, or the empty set $\emptyset$. It's easy to see $|\emptyset| = 0$. Conversely, if A contains infinitely many elements, A would be an *infinite* set and no such natural number k could describe the cardinality of A.

Different ways to describe a set can ultimately lead to the same set. For example, $\{2, 4, 6, 8\}$ and $\{x \mid x \text{ is even and } 0 < x < 10\}$ are different specifications but include exactly the same elements. Two sets A and B are equal, or $A = B$, iff they contain exactly the same elements. So $\{2, 4, 6, 8\} = \{x \mid x \text{ is even and } 0 < x < 10\}$. Phrasing the equality of two sets in another way, $A = B$ if two conditions holds: 1): for every x, if $x \in A$, then $x \in B$; 2): for every x, if $x \in B$, then $x \in A$. If every element of a set A is an element of a set B, then A is a *subset* of B, denoted as $A \subseteq B$. Thus, we see proving $A = B$ is equivalent as proving $1) : A \subseteq B$ and $2) : B \subseteq A$. Typically, to prove that two sets are equal requires proving the subset relations in both directions. Moreover, based on the definition of the subset relation, a set can be a subset of itself. If A is a subset of B but not equal to B, A is said to be a *proper subset* of B, written as $A \subset B$. Moreover, the usual operations of union, intersection, difference, Cartesian product and powerset are defined below.

$A \cup B =_{def} \{x \mid x \in A \text{ or } x \in B\}$	union
$A \cap B =_{def} \{x \mid x \in A \text{ and } x \in B\}$	intersection
$A - B =_{def} \{x \mid x \in A \text{ and } x \notin B\}$	difference
$A \times B =_{def} \{\langle a, b\rangle \mid a \in A \text{ and } b \in B\}$	Cartesian product
$\mathcal{P}(A) =_{def} \{X \mid X \subseteq A\}$	powerset

Relations and functions A *binary relation* on sets A and B is a subset of $A \times B$. A function f from its domain A to its co-domain B written as $f : A \to B$ is a relation $f \subseteq A \times B$ if $\langle a, b\rangle \in f$, then there is no $b' \in B$ distinct from b such that $\langle a, b'\rangle \in f$. A *total* function gets every element in its domain A mapped to something in B.

Alphabets, strings, languages An *alphabet* is a non-empty finite set of symbols, denoted by Σ. A *string* over an alphabet Σ is a finite sequence of symbols from Σ. The *length* of a string w ($|w|$) is the number of symbols in w. The empty string λ contains zero symbols and thus $|\lambda| = 0$.

A *language* or a *stringset* is a set of strings. As usual, Σ^* denotes the language which includes all possible strings over the alphabet Σ. Σ^n is the language with all possible strings of length n. $\Sigma^{\leq n}$ is the language that include all possible strings of length less than or equal to n. For example, if $\Sigma = \{0, 1\}$, $\Sigma^2 = \{00, 11, 01, 10\}$, $\Sigma^{\leq 2} = \{\epsilon, 0, 1, 00, 11, 01, 10\}$ and $\Sigma^* = \{\epsilon, 0, 1, 00, 11, 01, 10, 000 \ldots\}$.