IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Lecture notes – Formal languages

Langages et logique – ELU 610

This document contains all the mathematical elements needed by the course. We will discuss formal languages, syntax, semantics, and reasoning. The aim is to provide you with right background to understand the basics of languages and logics. In this context formal means *characterized mathematically*. It often helps because then the reasoning relies on the form not on the sense. Computation (preferably automated) can replace the subjectivity of human judgment and intuition.

Formal approaches are needed to characterize the fundamental concepts of a domain and to provide simple, precise and concise way to describe models. We use them to study foundations of languages and their properties but also to verify properties and to implement safe mechanisms of computation.

Formal approaches require discipline as they require the clarification of all assumptions and reasoning by using only explicitly and strictly a small set of (inference) rules.

Contents

1 Formal languages

Suppose we have a set A called the alphabet whose elements are called the symbols. A string (or word) over A is a (finite) sequence[1] of symbols of A. Its length is the number of elements of the sequence, it is a natural number as the sequences are required to be finite. The set of strings (over A) of length n is denoted A^n. The length of a string s is denoted $|s|$. Any element of A^n can be written $a_1a_2...a_n$ where each a_i is an element of A. As usual, $a_1a_2...a_n$ where for all a_i are equals to a a is denoted a^n. A specific string ϵ represents an empty sequence. It is the only element of the set A^0. The set of strings (over A) is defined by $\bigcup_{n\in\mathbb{N}} A^n$ and is denoted A^*. This set is countably infinite[2] (there is a bijective function between A^* and $\mathbb{N}$).

On A^*, one may define the concatenation operator $\cdot$. It takes two strings $s_1 = a_1a_2...a_{n_1}$ and $s_2 = b_1b_2...b_{n_2}$ and produces the string $s_1 \cdot s_2 = a_1a_2...a_{n_1}b_1b_2...b_{n_2}$.

Theorem 1

The set A^* equipped with the operator $\cdot$ and the element ϵ, denoted $(A^*, \cdot, \epsilon)$, forms a monoid: $\cdot$ is associative $(s_1 \cdot s_2) \cdot s_3 = s_1 \cdot (s_2 \cdot s_3)$ and ϵ is a neutral element for $\cdot$, $\epsilon \cdot s = s \cdot \epsilon = \epsilon$.

A^* is called the free monoid over A.

Definition 1, Formal language

A formal language $\mathcal{L}$ over an alphabet A is a subset of the free monoid over A, $\mathcal{L} \subset A^*$.

As such languages are sets, therefore, you can use the usual set operators on languages (union, intersection, complement). We can also reuse the mathematical practice of defining sets by extension (by the list of its elements) or intensionally (by a property of its elements). For example, if $A = \{\mathtt{a}, \mathtt{b}\}$:

- $\mathcal{L}_1 = \{\mathtt{a}, \mathtt{b}, \mathtt{aa}, \mathtt{aab}\}$
- $\mathcal{L}_2 = \{s \in A^* \mid |s| \text{ is even}\}$
- $\mathcal{L}_3 = \{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}\}$

[1] an ordered set
[2] Following the axiom of choice, it is possible to prove that any countable union of finite sets is countable.

- $\mathcal{L}_4 = \{s \in A^* \mid |s|_{\mathtt{a}} = |s|_{\mathtt{b}}\}$ where for any x of A and s of A^*, $|s|_x$ is the number of occurences of x in s
- $\mathcal{L}_5 = \{\epsilon\}$
- $\mathcal{L}_6 = \varnothing$, the empty language (notice that it is different from $\mathcal{L}_5$)
- $\mathcal{L}_7 = \mathcal{L}_1 \cdot \mathcal{L}_2 = \{s \in A^* \mid s = s_1 \cdot s_2 \wedge s_1 \in \mathcal{L}_1 \wedge s_2 \in \mathcal{L}_2\}$, this construction defines the concatenation of languages
- $\mathcal{L}_8 = \mathcal{L}_1^n = \{s \in A^* \mid s = s_1 \cdot ... \cdot s_n \wedge \{s_1, ..., s_n\} \subset \mathcal{L}_1^n\}$, we define $\mathcal{L}^0$ to be $\{\epsilon\}$
- $\mathcal{L}_9 = \mathcal{L}_1^* = \bigcup_{n\in\mathbb{N}} \mathcal{L}_1^n$, this defines the Kleene closure

Remark. For any alphabet A, there exists a non countable infinite number of languages over A. This result come from the Cantor theorem[3]. That's a lot of languages...

Computer science focuses on *finitely generated* languages. In these languages, the alphabet is finite and the language can be described by a finite set of information. But notice that they can still be an infinite number of strings in the language.

The study and classification of languages is the *formal language theory* [1, 4]. It focuses on how to define languages and (efficiently) recognize wether a string belongs to a language.

Consult `http://en.wikipedia.org/wiki/Formal_language`.

2 Regular expressions

Regular expression provides an easy to use language to specify formal languages.

Definition 2, Regular expression

A regular expression (RE) over an alphabet A is a well-formed[a] non empty string over $A \cup \{\varnothing, \epsilon, (,), |, *\}$ (we suppose that A does not already contain the added symbols). Here, well-formed means that opening and closing parentheses must be matched, $|$ is n-ary infix operator (it appears between each operand) and $*$ is unary postfix one (it appears after its operand).

[a]We will see later ways to formalize it.

Remark. It is important to notice the overloading of $\varnothing$ and ϵ that are used both for their string meaning and as a special symbol for building RE (where they will be in blue).

For example, $(a_1|a_2|...|a_n)*$ where $\{a_1, ..., a_n\} \subset A$ is a RE whereas $)|)$ is not.

We can associate a language over A to each RE r over A denoted $\mathcal{L}(r)$ and defined by:

1. $\mathcal{L}(\varnothing) = \varnothing$
2. $\mathcal{L}(\epsilon) = \{\epsilon\}$
3. $\mathcal{L}(a) = \{a\}$ for $a \in A$
4. $\mathcal{L}((r)) = \mathcal{L}(r)$
5. $\mathcal{L}(r_1 \cdot r_2) = \mathcal{L}(r_1) \cdot \mathcal{L}(r_2)$ for any r_1 and r_2
6. $\mathcal{L}(r_1|...|r_n) = \mathcal{L}(r_1) \cup ... \cup \mathcal{L}(r_n)$ for any r_1, ..., r_n $(n \geq 2)$
7. $\mathcal{L}(r*) = \mathcal{L}(r)^*$

[3]The cardinal of any set is strictly less that the one of the set of its subsets.

For example:

$$\begin{aligned}\mathcal{L}((a_1|\ldots|a_n)*) &= \mathcal{L}((a_1|\ldots|a_n))^* = \mathcal{L}(a_1|\ldots|a_n)^* = (\mathcal{L}(a_1) \cup \ldots \cup \mathcal{L}(a_n))^* \\ &= (\{a_1\} \cup \ldots \cup \{a_n\})^* = \{a_1, \ldots, a_n\}^*\end{aligned}$$

We can extend the language of RE with the following symbols: +, ? and any integer all being postfix unary operators defined as follows:

8. $r+ = r \cdot r* = r* \cdot r$
9. $r? = \epsilon | r$
10. $r0 = \epsilon$
11. $rk+1 = r \cdot rk$

It does change the set of RE but makes it easier to write some RE.

Definition 3, Regular language

A language $\mathcal{L}$ is regular if and only if there exists a RE r such that $\mathcal{L} = \mathcal{L}(r)$.

Defining language by regular expression is efficient but limited. There exists a lot of interesting languages that are not regular! For example, $\{a^n b^n, n > 0\}$ is not regular and it is useful for example to ensure that there is as much closing parentheses as opening ones in a regular expression (here $a =$ (and $b =$)).

We will see in the compilation part of the course that regular expressions fit well to define lexical analysis that produces tokens from raw symbol strings.

3 Finite automata

Finite automata provide efficient way to check whether a string pertain to a given language.

Definition 4, Finite automaton

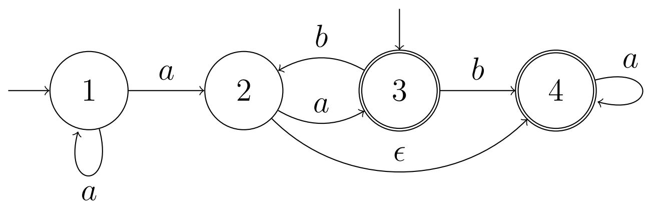
A finite automaton (FA) is a quintuple $(\Sigma, Q, \Delta, I, F)$ where:

- Σ is the alphabet, it is finite
- Q is the set of states, it is finite
- $\Delta \subset Q \times (\Sigma \cup \{\epsilon\}) \times Q$ is the transition relation, $(q, x, q') \in \Delta$ is denoted $q \xrightarrow[\Delta]{x} q'$[a]
- $I \subset Q$ is the set of initial states
- $F \subset Q$ is the set of final states

Sometimes, such an automaton is called ϵ-NFA (for non-deterministic).

[a]The Δ under the arrow may be dropped if it can be deduced from the context.

An automaton may be represented graphically as a graph where the vertices represent the states and the edges, the transitions. Vertices representing an initial (resp. final) state are marked with an incoming edge without label (resp. double circle). The left part of the figure 1 illustrates this representation called a transition diagram.



	a	b	ϵ
$\rightarrow 1$	$1, 2$		
2	3		4
$\rightarrow \underline{3}$		$2, 4$	
$\underline{4}$	4		

Figure 1: Transition diagram and transition table of FA

The automaton may also be defined by its transition table as in the right part of figure 1, where lines represent states and columns transitions on a symbol. The first column contains the list of states, the final ones are underlined and the initial ones are marked by an arrow.

Remark. Again, ϵ is overloaded both for the empty string and as a special symbol for *empty* transition (called ϵ-transition). The reuse of the same symbol aims at stressing their similarity of meaning because an ϵ-transition does not "read" anything.

A state q such that $\{q' \in Q \mid \exists x \in \Sigma, \Delta(q, x, q')\} \subset \{q\}$ is called a sink. If this set, sometimes called $next(q)$, is empty the state is called stuck.

Definition 5

Two transitions $q_1 \xrightarrow{x_1} q_1'$ and $q_2 \xrightarrow{x_2} q_2'$ are *consecutive* iff[a] $q_1' = q_2$. A path is a finite sequence of consecutive transitions. A path is *initial* (resp. *final*) if it starts (resp. ends) in an initial (resp. final) state. It is accepting, iff it is both initial and final. A state q' is accessible from a state q iff there exists a path from q to q'.

[a] *iff* means *if and only if*

When we do not care about the intermediary states, a path $q_0 \xrightarrow{x_1} q_1, ..., q_{n-1} \xrightarrow{x_n} q_n$ is denoted $q_0 \xrightarrow{x_1...x_n} q_n$. Keep in mind that $x_1...x_n$ is a string on $\Sigma \cup \{\epsilon\}$, it is called the *label* of the path.

If s is an element of $(\Sigma \cup \{\epsilon\})^*$, it can be written either ϵ^{k_0} or $\epsilon^{k_0} a_1 \epsilon^{k_1} ... a_n \epsilon^{k_n}$ with $\{a_1, ..., a_n\} \subset \Sigma$ and $\forall i \in [\![0..n]\!], k_i \in \mathbb{N}$. We define $s_{|_\Sigma}$ to be ϵ in the first case and the subsequence $a_1...a_n$ of s containing only the elements of Σ in the same order in the second case.

We write $q_0 \overset{s}{\Rightarrow} q_n$ when s is a string over Σ and there exists a path $q_0 \xrightarrow{s'} q_n$ where s' is a string over $\Sigma \cup \{\epsilon\}$ such that s $= s'_{|_\Sigma}$.

Definition 6

A string s of Σ^* is accepted by a FA $A = (\Sigma, Q, \Delta, I, F)$ if and only if there exists an accepting path labelled by s' with $s'_{|_\Sigma} = s$ otherwise it is rejected.

The language recognized by A, denoted $\mathcal{L}(A)$, is the set of its accepted strings. Two automata are *equivalent* iff they recognize the same language. It is denoted $\equiv_{\mathcal{L}}$.

Let's define the function accessible from $\wp(Q)$ to $\wp(Q)$ that associates a set of states to the set of the accessible states from its elements ($\mathsf{accessible}(P) = \{q' \in Q \mid \exists q \in P, \exists s \in \Sigma^*, q \xrightarrow{s} q'\}$). The dual notion is that of the language recognized when starting a state q: $\mathcal{L}_q = \mathcal{L}((\Sigma, Q, \Delta, \{q\}, F))$. Clearly, $\mathcal{L}_q$ is empty if there is no path from q to a final state.

Remark. Equivalent automaton may have very different form. For example, the empty automaton $A_\varnothing = (\Sigma, \varnothing, \varnothing, \varnothing, \varnothing)$[4] recognizes no string as any automaton $(\Sigma, Q, \Delta, I, F)$ where $\mathsf{accessible}(I) \cap F$ is empty[5] (Q and Δ may be large). They are all equivalent and recognize the empty language.

When using automaton to recognize strings, some states of the automaton may not be of interest. First, the states that cannot be reached from an initial state (it is not in $\mathsf{accessible}(I)$). Second any state q that has no accessible final state, *i.e.* $\mathsf{accessible}(\{q\}) \cap F = \varnothing$ (the string cannot be recognized anymore). An automaton is accessible (resp. coaccessible) iff all its states are accessible from an initial state (resp. have an accessible final state), *i.e.* $\mathsf{accessible}(I) = Q$ (resp. $\forall q \in Q, \mathsf{accessible}(\{q\}) \cap F \neq \varnothing$). It is a *trim* if it is both accessible and coaccessible. Intuitively, a trim does not contain any useless state.

Any FA $A = (\Sigma, Q, \Delta, I, F)$ can be *trimmed* to an automaton $\mathcal{T}(A)$ that is a trim. Let's define $Q_{a,c}$ as the subset of Q whose elements are accessible from an initial state and have an accessible final state, *i.e.* $Q_{a,c} = \{q \in \mathsf{accessible}(I) \mid \mathsf{accessible}(\{q\}) \cap F \neq \varnothing\}$. We define $\mathcal{T}(A)$ by $(\Sigma, Q_{a,c}, \Delta \cap (, Q_{a,c} \times \Sigma \times, Q_{a,c}), I \cap Q_{a,c}, F \cap Q_{a,c})$. It is easy to prove that $\mathcal{T}(A)$ is a trim and is equivalent to A (the accepting paths of A are exactly the accepting paths of $\mathcal{T}(A)$).

A FA $A = (\Sigma, Q, \Delta, I, F)$ is complete iff for all state q and symbol a, $\Delta(q, a) = \{q' \in Q \mid q \xrightarrow{a} q'\}$ is not empty. It is often convenient to complete a FA. Suppose that A is not complete. Let's define a new state $\bullet \notin Q$ and a new transition relation $\Delta^\bullet = \Delta \cup \{(q, a, \bullet) \mid a \in \Sigma \wedge \Delta(q, a) = \varnothing\} \cup \{(\bullet, a, \bullet) \mid a \in \Sigma\}$. The automaton $\mathcal{C}(A) = (\Sigma, Q \cup \{\bullet\}, \Delta^\bullet, I, F)$ is clearly complete and it is easy to prove that it is equivalent to A. If A is complete, we define $\mathcal{C}(A)$ to be A.

Notice that an automaton may be (1) a complete trim, (2) a trim not complete whose completion is not a trim and (3) complete but not a trim and trimming it removes completness:

$$\begin{cases} (1) & (\{a, b\}, \{q_0, q_1\}, \{q_0 \xrightarrow{a} q_1, q_0 \xrightarrow{b} q_0, q_1 \xrightarrow{a} q_1, q_1 \xrightarrow{b} q_0\}, \{q_0\}, \{q_1\}) \\ (2) & (\{a, b\}, \{q_0, q_1\}, \{q_0 \xrightarrow{a} q_1\}, \{q_0\}, \{q_1\}) \\ (3) & (\{a, b\}, \{q_0, q_1\}, \{q_0 \xrightarrow{a} q_1, q_0 \xrightarrow{b} q_0, q_1 \xrightarrow{a} q_1, q_1 \xrightarrow{b} q_1\}, \{q_0\}, \{q_0\}) \end{cases}$$

But, we can get a *small* complete automaton out of any FA by first trimming it and then completing it. The result is a quasi-trim where all states are accessible and all but possibly one states have an accessible final state (the $\bullet$ state if it is added by $\mathcal{C}$). More precisely, if the number of states of an automaton A is denoted $|A|$, then $|A| \leq |\mathcal{C}(\mathcal{T}(A))| \leq |A| + 1$.

Definition 7, Recognizable language

A language $\mathcal{L}$ is recognizable if and only if there exists a FA A such that $\mathcal{L} = \mathcal{L}(A)$.

For any FA $A = (\Sigma, Q, \Delta, I, F)$ let's define the FA $\mathcal{I}(A) = (\Sigma, Q \cup \{q_0\}, \Delta', \{q_0\}, F)$ where $q_0 \notin Q$ and the transition function is extended by transitions ϵ from q_0 to all elements of I, *i.e.* $\Delta' = \Delta \cup \{(q_0, \epsilon, x) \mid x \in I\}$. It is easy to prove that $A \equiv_{\mathcal{L}} \mathcal{I}(A)$. Some authors constrain the definition of FA to have a unique initial state. It can also be used to suppose that there exists an initial state.

[4]Some authors do not accept such empty automaton while adding them do not add complexity...

[5]In particular, this is true if either F or I are empty.

Definition 8, Deterministic finite automaton

A FA $(\Sigma, Q, \Delta, I, F)$ is deterministic (DFA) iff:

1. I is a singleton
2. there is no transition on the empty string, $\{q \in Q \mid \exists q' \in Q, q \xrightarrow{\epsilon} q'\} = \varnothing$
3. for all $q \in Q$ and all $a \in \Sigma$, $\{q' \in Q \mid q \xrightarrow{a} q'\}$ is a singleton

The DFA is denoted $(\Sigma, Q, \delta, q_0, F)$ with $I = \{q_0\}$, $\forall q \in Q, a \in \Sigma, \{q' \in Q \mid q \xrightarrow{a} q'\} = \{\delta(q, a)\}$.

Notice that a DFA cannot have an empty set of states. So trimming a DFA may lead to the empty FA which is not considered a DFA. This happens when A has no accessible final state.

Theorem 2

For every finite non empty automaton A there exists an equivalent deterministic automaton $\mathcal{D}(A)$.

Proof. If $A = (\Sigma, Q, \Delta, I, F)$, let's define $\mathcal{D}(A) = (\Sigma, \wp(Q), \delta, I_\mathcal{D}, F_\mathcal{D})$ with

$$\begin{cases} I_\mathcal{D} = I \cup \{q \in Q \mid \exists i \in I, n \geq 1, i \xrightarrow{\epsilon^n} q\} \\ F_\mathcal{D} = \{P \subset Q \mid P \cap F \neq \varnothing\} \\ \delta(P, a) = \{q \in Q \mid \exists p \in P, p \overset{a}{\Rightarrow} q\} \end{cases}$$

$\mathcal{D}(A)$ is clearly deterministic. Proving it recognizes the same language as A must be done by double inclusion.

- $\mathcal{L}(A) \subset \mathcal{L}(\mathcal{D}(A))$: if ϵ is in $\mathcal{L}(A)$ then an ϵ-transition goes from an initial state to a final state hence a final state is in $I_\mathcal{D}$ making it final for A'. Suppose $s = a_1 ... a_n$ is recognized by A, there exists an accepting path $q_0 \xRightarrow{a_1} q_1 ... q_{n-1} \xRightarrow{a_n} q_n$. Then q_0 is in I. Let's show by induction on n that there also exists a path $I \xrightarrow[\delta]{a_1} P_1 ... P_{n-1} \xrightarrow[\delta]{a_n} P_n$ in $D(A)$. $(n = 1)$ as $q_0 \xRightarrow{a_1} q_1$, $I \xrightarrow[\delta]{a_1} P_1$ where P_1 is $\delta(I, a_1) = \{q \in Q \mid \exists p \in I, p \xRightarrow{a_1} q\}$ that contains at least q_1. $(n-1 \Rightarrow n)$ suppose $I \xrightarrow[\delta]{a_1} P_1 ... P_{n-2} \xrightarrow[\delta]{a_{n-1}} P_{n-1}$ exists with $q_{n-1} \in P_{n-1}$, as $q_{n-1} \xRightarrow{a_n} q_n$, $P_{n-1} \xrightarrow{a_n} P_n$ where P_n is $\delta(P_{n-1}, a_n) = \{q \in Q \mid \exists p \in P_{n-1}, p \xRightarrow{a_n} q\}$ that contains at least q_n. Furthermore, as q_n is final, $P_n \cap F$ is not empty which makes P_n final in $\mathcal{D}(A)$ meaning s is accepted by $\mathcal{D}(A)$.
- $\mathcal{L}(\mathcal{D}(A)) \subset \mathcal{L}(A)$ conversely if ϵ is in $\mathcal{L}(\mathcal{D}(A))$ then $I_\mathcal{D} \in F_\mathcal{D}$ meaning there exists a q in $I_\mathcal{D} \cap F$, so there exists $i \in I$ such that $i \overset{\epsilon}{\Rightarrow} q$ meaning $\epsilon \in \mathcal{L}(A)$. Suppose $s = a_1 ... a_n$ is recognized by $\mathcal{D}(A)$ there exists an accepting path $P_0 \xrightarrow[\delta]{a_1} P_1 ... P_{n-1} \xrightarrow[\delta]{a_n} P_n$ $(P_0 = I)$. By definition $P_n \cap F$ is not empty, let's choose q_n in it. As $P_{n-1} \xrightarrow[\delta]{a_n} P_n$, $P_n = \{q \in Q \mid \exists p \in P_{n-1}, p \xRightarrow{a_n} q\}$ which means that there exists a $q_{n-1} \in P_{n-1}$ such that $q_{n-1} \xRightarrow{a_n} q_n$ $(q_{n-1} \xrightarrow[\delta]{a_n} q_n)$. The construction can be repeated until q_0, giving us a path $q_0 \xRightarrow{a_1 ... a_n} q_n$ where q_n is in F and q_0 is in I.

□

The definition of $\mathcal{D}(A)$ in the proof is called the *subset construction*, it gives an algorithm to determinize a NFA. It has been proved that this process of determinization is optimal. Notice that the resulting DFA can be much larger than the FA. Indeed, the size of the power set of a finite set of size n is 2^n. The complexity of the determinization of A is therefore $\mathcal{O}(2^{|A|})$.

The two kinds of automaton are useful. Producing a DFA is hard (typically the construction is in $\mathcal{O}(|A|^3)$ but may in the worst case reach $\mathcal{O}(|A|^2 2^{|A|})$) and often results in a large automaton but gives an efficient way of recognizing a strings s in $\mathcal{O}(|s|)$ (there is a unique path). Producing a NFA is much easier (construction in $\mathcal{O}(|A|)$) and the automaton is much smaller but using it for recognizing strings is much less efficient as one has to follow all paths resulting in a complexity of $\mathcal{O}(|s| \times |A|)$. So if an automaton is going to be used several times to recognize strings the cost of its determinization can be amortized.

A string s *distinguish* two states q_1 and q_2 of a DFA iff either there is a path from q_1 labeled s to a final state and that is not true for q_2 or the situation is reversed. Two states are *distinguishable* if there exists a string that distinguish them.

The *Nerode* equivalence is defined as the inverse of distinguishability. Let q_1 and q_2 be two states of a DFA, $q_1 \sim q_2$ if and only if for all strings s, $q_1 \xrightarrow{s} q_1'$ and $q_2 \xrightarrow{s} q_2'$ then $q_1' \in F \Leftrightarrow q_2' \in F$. Clearly, $q_1 \sim q_2$ is equivalent to $\mathcal{L}_{q_1} = \mathcal{L}_{q_2}$ and if $q_1 \sim q_2$ then q_1 is final iff q_2 is final (both language either contains the empty string or not). If furthermore the state q is accessible ($q_0 \xrightarrow{s_q} q$) then $\mathcal{L}_q = \{s \in \Sigma^* \mid s_q \cdot s \in \mathcal{L}\}$[6]. Lastly, if $q_1 \sim q_2$, for all string s, either there exists q_1' such that $q_1 \xrightarrow{s} q_1'$ and there must be a q_2' with $q_2 \xrightarrow{s} q_2'$ and $q_1' \sim q_2'$ or both $q_1 \not\xrightarrow{s}$ and $q_2 \not\xrightarrow{s}$ (otherwise there would be a string starting by s distinguishing q_1 and q_2).

Suppose $A = (\Sigma, Q, \delta, q_0, F)$, let's define $M(A) = (Q/\sim, \delta^\sim, [q_0]_\sim, F/\sim)$ with $\delta^\sim([q]_\sim, a) = [\delta(q, a)]_\sim$. $\delta^\sim$ is a well defined function because if $[q_1]_\sim = [q_2]_\sim$ then $q_1 \sim q_2$ and following a must give the same result (see above) so either $\delta(q_1, a)$ exists and then $\delta(q_1, a) \sim \delta(q_2, a)$ or both do not exists and $\delta^\sim$ is also undefined. Notice that if the DFA is complete then $\delta^\sim$ is always defined and $M(A)$ is therefore complete.

Theorem 3

Let A be a trim DFA then $M(A)$ is a *minimal*[a] automaton equivalent to A.

[a]It has less state than any other equivalent automata.

You can consult [4, chapter 4, section 4.4] for a proof.

It is possible to combine the various transformations presented above. The usual chaining being $\mathcal{C}(\mathcal{M}(\mathcal{T}(\mathcal{D}(A))))$ to get a complete minimal DFA. Notice that it may not be minimal if the last transformation needs to add a sink state.

Theorem 4, Kleene

The regular class of language is exactly the recognizable class.

The proof is done in [4, chapter 3] by providing algorithm to build a FA out of a RE and conversely to produce a RE from a DFA. The idea behind RE to FA is summarized in figure 2 it relies on an induction on the structure of RE.

[6]Sometimes, denoted $s_q^{-1}\mathcal{L}$.

[7]We reproduce here the original automaton but we could also use $A_\varnothing$...

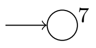
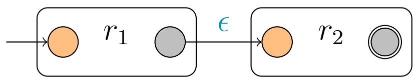
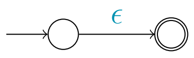
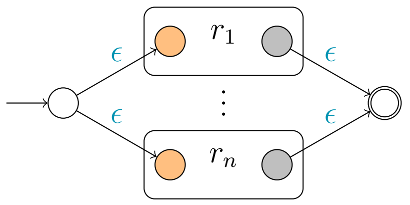
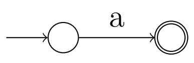
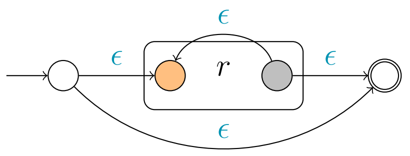
base RE	FA	combined RE	FA (🟠 initial, ● final in sub-FA)
$\varnothing$		$r_1 \cdot r_2$	
ϵ		$r_1 \mid \ldots \mid r_n$	 r_1 ϵ ϵ ϵ ϵ r_n
a		$r*$	

Figure 2: Producing a FA from a RE

The theorem means that it is easier to define a language using regular expressions and then convert it to an automaton.

Theorem 5

The regular language class is closed under union, intersection, complement, difference, star and concatenation (*i.e.* any combination of regular languages using one of these operators is a regular language).

You can find all the proof in [4, chapter 4].

Theorem 6

Any finite language is regular.

Proof. Since regular languages are closed under union, it is sufficient to prove that any singleton is regular. Suppose our alphabet is Σ, if $\mathcal{L} = \{s\}$, with $s = a_1 ... a_n \in \Sigma^*$, it is easy to see that it is recognized by the DFA $(\Sigma, \{q_0, ..., q_n\}, \{q_0 \xrightarrow{a_1} q_1, ..., q_{n-1} \xrightarrow{a_n} q_n\}, q_0, \{q_n\})$. $\square$

The theorems 4 and 5 provides a concrete efficient mean of describing languages. For example, if $\mathcal{L}_{int} =$, we can define a language $\mathcal{L}$ as $\mathcal{L}(\texttt{(0|...|9)*}) \cup \mathcal{L}(\texttt{(0|...|9)*.(0|...|9)*}) \cup \mathcal{L}(\texttt{(+|-|*|/)*})$.

4 Grammar

Definition 9, Grammar

A grammar is a quadruple (Σ, V, P, S) where:

- Σ is the alphabet of symbols called the terminals, it is finite
- V is the set of variables called the non-terminals, it is finite and $\Sigma \cap V = \varnothing$

Class	Name	Production form	Recognizer
Type-0	Recursively enumerable	no restriction	Turing Machine (TM)
Type-1	Context-sensitive (CSG)	$\alpha A\beta \to \alpha\gamma\beta$ $A \in V$, $\alpha, \beta, \gamma \in (\Sigma \cup V)^*$	Linear-bounded TM
Type-2	Context-free (CFG)	$A \to \gamma$ $A \in V$, $\gamma \in (\Sigma \cup V)^*$	Pushdown Automaton (PDA)
Type-3	(Left) Regular	$A \to \gamma$ $A \in V$, $\gamma \in (V \cdot \Sigma^*) \cup \Sigma^*$	Finite Automaton
	(Right) Regular	$A \to \gamma$ $A \in V$, $\gamma \in (\Sigma^* \cdot V) \cup \Sigma^*$	

Figure 3: chomsky hierarchy

- $P \subset (\Sigma \cup V)^* \cdot V \cdot (\Sigma \cup V)^* \times (\Sigma \cup V)^*$ is the set of productions, (α, β) is written $\alpha \to \beta$
- $S \in V$ is the axiom

The definition relies on the string concatenation operator but to simplify the rules the $\cdot$ operator will be represented the same way than the symbol sequence, $s_1 \cdot s_2$ is written $s_1 s_2$.

Definition 10, Derivation

Direct derivation of a grammar (Σ, V, P, S) is a relation on $(\Sigma \cup V)^*$ denoted $s_1 \to s_2$ defined when there exists a production $\alpha \to \beta$ in P such that $s_1 = x\alpha y$ and $s_2 = x\beta y$ for any strings x and y.

The relation *derives* is defined as the transitive closure of direct derivation. It is denoted, as usual, $\to^*$.

A direct derivation $x\alpha y \to x\beta y$ is leftmost (resp. rightmost) if it is the leftmost (resp. rightmost) non-terminal that is replaced: x (resp. y) contains only terminals. It is denoted $\xrightarrow[lm]{}$ (resp. $\xrightarrow[rm]{}$).

There can be derivations that are neither leftmost nor rightmost.

Definition 11

The language generated by a grammar $G = (\Sigma, V, P, S)$ is the set of strings that be derived from the axiom: $\mathcal{L}(G) = \{s \in \Sigma^* \mid S \to^* s\}$. Similarly to automata, grammars are equivalent when they generate the same language, it is written using the same symbol $\equiv_{\mathcal{L}}$.

The various grammars can be classified using their *complexity*, it is the chomsky hierarchy presented in the table of figure 3. It is easy to see that Type-3 $\subsetneq$ Type-2 $\subsetneq$ Type-1 $\subsetneq$ Type-0.

Definition 12, Derivation tree

Let G be a grammar (Σ, V, P, S). A derivation tree of G is a finite tree satisfying:

1. each leaf is labeled by a terminal (from $\Sigma \cup \{\epsilon\}$),
2. each other node is labeled by a non-terminal (from V),
3. if a node, labeled B, has n children x_1, ..., x_n then a production $B \to x_1 ... x_n$ must belong to P.

If the root of a derivation tree is A and its leafs ordered from left to right are a_1, ..., a_n (it is called the *yield*) then it corresponds to the derivation $A \to^* a_1 ... a_n$.

Theorem 7

For every derivation tree, there is a unique leftmost and a unique rightmost derivation.

Proof. The proof is done by an induction on the height of the derivation tree. We give the proof for leftmost, the proof for rightmost is identical replacing left by right. $(h = 1)$ the tree is built a unique production $A \to a_1 ... a_n$ and as A is unique non-terminal of A then $A \underset{lm}{\to} a_1 ... a_n$ and it is the only leftmost derivation. (h=n) it is easy to see that the derivation tree must be of the following form $A \to \triangle_1 \dots \triangle_k$ where for each $i \in [\![1..k]\!]$, $\triangle_i$ is a derivation tree of height strictly smaller that n whose root is X_i and yields s_i. By induction hypothesis, there exists a unique leftmost derivation for each of these sub-trees $(X_i \underset{lm}{\to}^* s_i)$. Thus $A \underset{lm}{\to} X_1 ... X_k \underset{lm}{\to}^* s_1 X_2 ... X_k \underset{lm}{\to}^* \dots \underset{lm}{\to}^* s_1 ... s_k$ is the unique leftmost derivation. □

Definition 13, Ambiguous

For a grammar, a string that can be generated by several (different) derivation trees is ambiguous. A grammar with an ambiguous string is also said to be ambiguous. A language that cannot be described by an unambiguous grammar is said to be *inherently ambiguous*. The set of language having an unambiguous grammar is denoted $\mathbb{U}$.

For example, the grammar $(\{\texttt{1}, \texttt{+}\}, \{A, S\}, \{S \to A, A \to A\texttt{+}A \mid \texttt{1}\}, S)$ generates ambiguously `1+1+1` as either (`1+1`)`+1` or `1+`(`1+1`).

Often, we gave remove ambiguity changing but some grammars are inherently ambiguous meaning that we cannot get rid of the ambiguities. For example, any grammar for the language $\{\texttt{a}^i\texttt{b}^j\texttt{c}^k \mid i = j \vee j = k\}$ is ambiguous. Intuitively, one derivation tree is obtained by "counting" the `a` and `b` while another by "counting" the `b` and `c`.

4.1 Regular grammars

Theorem 8

A right regular grammar is equivalent to a left regular grammar.

Proof. Suppose $G = (\Sigma, V, P, S)$ is a right regular grammar. If the axiom S appear in the right hand side of a production, we can transform G in $(\Sigma, V \cup \{S_0\}, P \cup \{S_0 \to S\}, S_0)$ with $S_0 \notin V$. So we can suppose that S does not appear in the right hand side of a production.

Then we define the grammar $G' = (\Sigma, V, P', S)$ with

$$\begin{aligned} P' = &\{B \to Ax \mid A \to xB \in P, A \in V \setminus \{S\}, x \in \Sigma^*, B \in V\} \\ &\cup \{B \to x \mid S \to xB \in P, x \in \Sigma^*, B \in V\} \\ &\cup \{S \to Ax \mid A \to x \in P, A \in V \setminus \{S\}, x \in \Sigma^*\} \\ &\cup \{S \to x \mid S \to x \in P, x \in \Sigma^*\} \end{aligned}$$

G' is left regular and is equivalent to G. Indeed a derivation for the string $s \in \Sigma^*$ has the form $S = A_0 \to x_1A_1 \to^* x_1...x_nA_n = s$ with $A_i \to x_{i+1}A_{i+1}$ and $A_{n-1} \to x_n$ productions of G such that $A_i \neq S$ for $i > 0$. This derivation is possible if and only if a derivation of G' is $S = A_n \to A_{n-1}x_n \to^* A_1x_2...x_{n-1}x_n \to x_1...x_n = s$ with the productions $A_1 \to x_1$ and $A_i \to A_{i-1}x_i$ for $i > 1$. □

Theorem 9

The Type-3 class of language generated by a regular grammar is exactly the regular class (recognized by FA).

Proof. Suppose $G = (\Sigma, V, P, S)$ is a right regular grammar (it is general enough by theorem 8). We can build $G' = (\Sigma, V', P', S)$ an equivalent grammar such that for every production $A \to x$ or $A \to xB$ of P', $|x| = 1$[8]. Then the following FA recognizes $\mathcal{L}(G)$:

$$A = (\Sigma, V \cup \{q\}, \{A \xrightarrow{a} B \mid A \to aB \in P\} \cup \{A \xrightarrow{a} q \mid A \to a \in P\}, S, \{q\})$$

Conversely, suppose $A = (\Sigma, Q, \delta, q_0, F)$ is a FA. The following grammar is right regular and generates $\mathcal{L}(A)$.

$$G = (\Sigma, Q, \{q \to aq' \mid q \xrightarrow{a} q' \in \delta\} \cup \{q \to \epsilon \mid q \in F\}, q_0)$$

□

4.2 Context-free grammars (CFG)

The previously seen language $\{a^nb^n, n > 0\}$ is not regular but is context-free (CF). It can be generated by the following CFG $(\{a, b\}, \{S, A\}, \{S \to A, A \to aAb, A \to ab\}, S)$. Notice that a CF language can only *count two things*. For example, $\{a^nb^nc^n \mid n > 0\}$ is not CF.

Theorem 10

The Type-2 class of language generated by a CF grammar is closed by union, concatenation and star.

Proof. Suppose that G_1 and G_2 are defined by $G_1 = (\Sigma_1, V_1, P_1, S_1)$ and $G_2 = (\Sigma_2, V_2, P_2, S_2)$.

- $G = (\Sigma_1 \cup \Sigma_2, V_1 \cup V_2 \cup \{S\}, P_1 \cup P_2 \cup \{S \to S_1, S \to S_2\}, S)$ generates $\mathcal{L}(G_1) \cup \mathcal{L}(G_2)$.

[8]The idea is to replace any production $A \to x_1...x_n$ with $n \geq 2$ using the following set of productions $\{A \to [x_1][x_2...x_m], [x_2...x_m] \to [x_2][x_3...x_m], ..., [x_{m-1}x_m] \to [x_{m-1}][x_m]\} \cup \{[x_i] \to x_i \mid 1 \leq i \leq n\}$.

- $G = (\Sigma_1 \cup \Sigma_2, V_1 \cup V_2 \cup \{S\}, P_1 \cup P_2 \cup \{S \to S_1 \cdot S_2\}, S)$ generates $\mathcal{L}(G_1) \cdot \mathcal{L}(G_2)$.
- $G = (\Sigma_1, V_1 \cup \{S\}, P_1 \cup \{S \to SS_1, S \to \epsilon\}, S)$ generates $\mathcal{L}(G_1)^*$.

□

Definition 14, Pushdown automaton

A pushdown automaton (PDA) is a tuple $A = (\Sigma, \Gamma, Q, \Delta, I, Z, F)$ where:

- Σ is the input alphabet, it is finite
- Γ is the pushdown alphabet, it is finite and not empty
- Q is the set of states, it is finite
- $\Delta \subset Q \times \Gamma \times (\Sigma \cup \{\epsilon\}) \times Q \times \Gamma^*$ is the transition relation, the elements (q, z, x, q', γ) of Δ are written $q \xrightarrow{x,z/\gamma} q'$
- $I \subset Q$ is the set of initial states
- $Z \subset \Gamma$ is the set of initial pushdown symbols
- $F \subset Q$ is the set of final states

For $\{q, z, x\}$ from $Q \times \Gamma \times (\Sigma \cup \{\epsilon\})$, $\Delta(q, z, x)$ denotes the set $\{(q', \gamma) \in Q \times \Gamma^* \mid (q, z, x, q', \gamma) \in \Delta\}$.

Definition 15, Configuration

A configuration of a PDA is an element of $Q \times \Sigma^* \times \Gamma^*$ composed by a the current state of the PDA, the remaining string and the current stack.

The configuration $(q_2, s, \gamma_2\gamma_1)$ is a *direct successor* of the configuration $(q_1, xs, z\gamma_1)$, written $(q_1, xs, z\gamma_1) \vdash (q_2, s, \gamma_2\gamma_1)$ if $(q_2, \gamma_2) \in \Delta(q_1, z, x)$. The *successor* relation is the transitive closure of direct successor denoted $\overset{*}{\vdash}$.

The intuition here is that when doing a transition from a state q_1 to a state q_2, the PDA consumes a symbol x from the input and a symbol z from the stack and pushes back on the stack a sequence of pushdown symbols γ_2.

Definition 16

A string s of Σ^* can be accepted by a PDA $A = (\Sigma, \Gamma, Q, \Delta, I, Z, F)$:

- s is *accepted by final state* (FS) if and only if there exists a path from configuration (q_I, s, z_I) to (q_F, ϵ, γ) where $q_I \in I$, $z_I \in Z$, $q_F \in F$ and γ is any stack;
- s is *accepted by empty stack* (ES) if and only if there exists a path from configuration (q_I, s, z_I) to (q, ϵ, ϵ) where $q_I \in I$ and $z_I \in Z$.

The language recognized by A in final state (resp. by empty state), denoted $\mathcal{L}_{FS}(A)$ (resp.

$\mathcal{L}_{ES}(A)$), is the set of its accepted strings by final state (resp. by empty stack).

For example, the following PDA recognizes the language $\{a^n b^n \mid n > 0\}$ both by final state and empty stack.

$$\begin{aligned}&(\{a,b\},\{z,\perp\},\{q_0,q_1,q_2\},\\&\quad\{(q_0,\perp,a,q_0,z\perp),(q_0,z,a,q_0,zz),(q_0,z,b,q_1,\epsilon),(q_1,z,b,q_1,\epsilon),(q_1,\perp,\epsilon,q_2,\epsilon)\},\\&\quad\{q_0\},\{\perp\},\{q_2\})\end{aligned}$$

For example, *aabb* is recognized but not *aab*:

$$(q_0,aabb,\perp)\vdash(q_0,abb,z\perp)\vdash(q_0,bb,zz\perp)\vdash(q_1,b,z\perp)\vdash(q_1,\epsilon,\perp)\vdash(q_2,\epsilon,\epsilon)$$
$$(q_0,aab,\perp)\vdash(q_0,ab,z\perp)\vdash(q_0,b,zz\perp)\vdash(q_1,\epsilon,z\perp)\not\vdash$$

Theorem 11

For any PDA A, there exists another PDA A' such that $\mathcal{L}_{FS}(A) = \mathcal{L}_{ES}(A')$ (resp. $\mathcal{L}_{ES}(A) = \mathcal{L}_{FS}(A')$).

See [4, section 6.2] for a proof.

Theorem 12

The Type-2 class of languages generated by a context-free grammar is exactly the class of languages recognized by a PDA by empty stack (resp. final state).

See [4, section 6.3] for a proof.

Definition 17, Deterministic pushdown automaton

A PDA $A = (\Sigma, \Gamma, Q, \Delta, I, Z, F)$ is deterministic (DPDA) iff:

1. $|\Delta(q,z,x)| \leq 1$ for any $q \in Q$, $z \in \Gamma$ and $x \in (\Sigma \cup \{\epsilon\})$,
2. if $\Delta(q,z,\epsilon) \neq \varnothing$ then $\Delta(q,z,a) = \varnothing$ for all a of Σ.

A language is deterministic iff it is recognized by a DPDA by final state. Their set is denoted $\mathbb{D}$.

Theorem 13

A Type-3 language is deterministic: Type-3 $\subsetneq \mathbb{D}$.

Theorem 14

A deterministic language has an unambiguous CFG: $\mathbb{D} \subsetneq \mathbb{U}$.

See [4, section 6.4] for a proof of these two theorems.

The language $\{a^n b^n \mid n > 0\} \cup \{a^n b^{2n} \mid n > 0\}$ is in $\mathbb{U} \setminus \mathbb{D}$. It cannot be deterministic because we need to read at least as b as we have read a before deciding wether the b should be read by pairs. This requires an unbounded capacity not possible with DPDA. The language is unambiguous because the grammar $S \to aTb, T \to aTb, T \to \epsilon, S \to aUbb, U \to aUbb, U \to \epsilon$ is not ambiguous.

5 Term algebra

A term algebra is defined as an algebraic structure over a signature. A signature (Σ, ar) is a set Σ equipped with a (complete) function ar from Σ to $\mathbb{N}$. The elements c of Σ are called the constructors of the language and $ar(c)$ is called the arity of c. A constructor of arity 0 is called a constant.

In the rest of the document, we use a vector notation $\overrightarrow{x}$ to represent $x_1, ..., x_n$. The length n of the vector should be clear from the context.

Definition 18, Term algebra

A term algebra T_Σ over a signature (Σ, ar) is the smallest part of Σ^* such that:

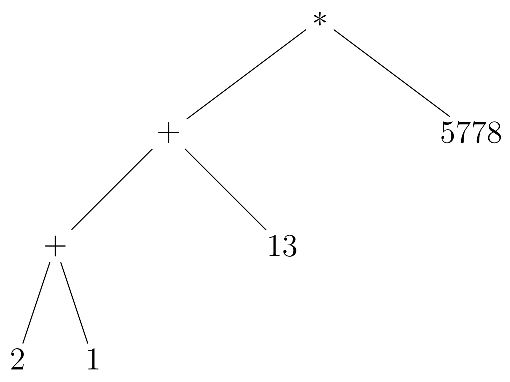
1. $\{c \in \Sigma \mid ar(c) = 0\} \subset T_\Sigma$
2. $c \in \Sigma \wedge ar(c) = n \geq 1 \wedge \overrightarrow{M} \in T_\Sigma^n \Rightarrow c(\overrightarrow{M}) \in T_\Sigma$

The element of a term algebra are called terms.

Notice that T_Σ is a language over Σ. Notice also that there is no constraint on the size of Σ.

For example, $\Sigma = \mathbb{N} \cup \{+, *\}$, $ar(+) = ar(*) = 2$ and $n \in \mathbb{N} \Rightarrow ar(n) = 0$ then T_Σ defines a simple expression language over integers. Its concretes syntax would probably requires the symbols $\{$`0`, `1`, `2`, ..., `9`, `+`, `*`, `(`, `)`$\}$.

Terms are trees where each constructor is a node and each of its direct sub-terms is a child.



Theorem 15, Structural induction

Let P be a property on terms such that:

1. P is true for all constants,
2. for each constructor c of Σ with $ar(c) = n$, supposing P is true for terms $M_1, \ldots, M_n$ we can prove that P is true for $c(M_1, \ldots, M_n)$.

Then P is true for all elements of T_Σ.

Proof. Let E be the set of terms where P is true. By hypothesis, E contains the constants and contains $c(M_1, \ldots, M_n)$ whenever it contains $M_1, \ldots, M_n$. By definition, T_Σ is the smallest of such sets meaning that $T_\Sigma \subset E$ and P is true for all T_Σ. □

This theorem gives a direct *method of proof* qualified as proof by (structural[9]) induction[10]: prove the property on constants (sometimes called the base cases) and then prove each constructor preserves it (sometimes called the inductive steps). For example, it is easy to prove by induction the following property.

Theorem 16

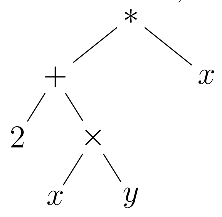
If $\Sigma_1 \subset \Sigma_2$ then $T_{\Sigma_1} \subset T_{\Sigma_2}$.

Proof. **base**: any constant of Σ_1 is in Σ_2 ; **induction step**: for $\{M_1, \dots, M_n\} \subset T_{\Sigma_1}$, let's suppose $\{M_1, \dots, M_n\} \subset T_{\Sigma_2}$, then by definition of T_{Σ_1}, $c(M_1, \dots, M_n) \in T_{\Sigma_1}$ and by definition of T_{Σ_2}, $c(M_1, \dots, M_n) \in T_{\Sigma_2}$ for all c of Σ_1 (they are also constructors of Σ_2). $\square$

It gives also a direct *method of definition*. To define a function on T_Σ, define it on constants and how it behave on constructors. For example, we can define the size of a term by:

$$\begin{cases} |c| = 0 \text{ if } ar(c) = 0 \\ |c(M_1, ..., M_{ar(c)})| = 1 + max_{i \in [\![1..ar(c)]\!]} |M_i| \text{ if } ar(c) \geq 1 \end{cases}$$

It is often necessary to give names to part of terms. We have already done it on several occasions, *e.g.* when talking about the "term" $c(M_1, ..., M_n)$. Formally, this requires to extends the set of symbols to include names. Suppose we have a signature (Σ, ar) and countable set of names $\mathcal{N}$ such that $\Sigma \cap \mathcal{N}$. We define terms with names over $(\Sigma, \mathcal{N}, ar)$ to be the elements of $T_{\Sigma \cup \mathcal{N}}$ with $ar(x) = 0$ for all $x \in \mathcal{N}$. The elements of $T_{\Sigma \cup \mathcal{N}}$ that does not contain names are ground terms (or closed terms), their set is exactly T_Σ. In the tree view, names are leafs as constants. For example:



It is possible to define by induction the set of names N of a term with names:

1. $N(c) = \varnothing$ for $c \in \Sigma$ and $ar(c) = 0$,
2. $N(x) = \varnothing$ for x in $\mathcal{N}$,
3. $N(c(M_1, ..., M_{ar(c)})) = \bigcup_{i \in [\![1..ar(c)]\!]} N(M_i)$ when $c \in \Sigma$ and $ar(c) \geq 1$.

It is possible to replace some names of a term by other terms using substitutions.

Definition 19, Substitution

A substitution is a function from $\mathcal{N}$ to $T_{\Sigma \cup \mathcal{N}}$ which is the identity function except for a finite subset of $\mathcal{N}$ called its domain. A substitution whose domain is $\{\vec{x}\}$ is denoted $[\overrightarrow{x \mapsto \sigma(x)}]$. A substitution σ operates on $T_{\Sigma \cup \mathcal{N}}$ by:

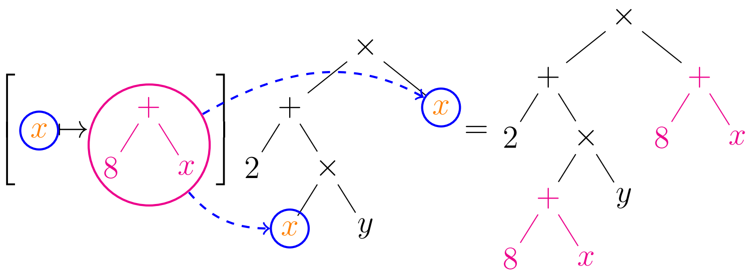
1. $\sigma x = \sigma(x)$, for x in $\mathcal{N}$,
2. $\sigma c = c$, for $c \in \Sigma$ and $ar(c) = 0$,

[9] There exists several form of induction but in our course, we will only cover structural induction hence we will only use the term induction.

[10] Induction is akin to usual *mathematical induction* on $\mathbb{N}$ (called recurrence in french).

3. $\sigma c(M_1, ..., M_n) = c(\sigma M_1, ..., \sigma M_n)$, for $c \in \Sigma$ and $ar(c) \geq 1$,

A substitution operate on a term by replacing **all** occurences of each of the substitution variables in the term. For example, using a tree view:



Remark. As illustrated in the previous example, as the substitution image may contain a name of its domain, the term may still contains the substituted names (x in the example).

Names are used in two ways: as metavariables or as variables. In the former case, the aim is to speak of a set of terms whereas in the latter, it is to speak of a yet unknown term. The set of names is then the union of $\mathcal{M}$, the set of metavariables, and $\mathcal{X}$, the set of variables.

When names are metavariables, they are generally written capitalized. The term algebra $T_{\Sigma \cup \mathcal{M} \cup \mathcal{X}}$ defines the language of metaterms whereas terms are elements of $T_{\Sigma \cup \mathcal{X}}$. For example, if c is a constructor of Σ of arity 2 and M_1 and M_2 are metavariables (elements of $\mathcal{M}$) then $c(M_1, M_2)$ is a metaterm. A metaterm containing at least one metavariable (it is an element from $T_{\Sigma \cup \mathcal{M} \cup \mathcal{X}} \backslash T_{\Sigma \cup \mathcal{X}}$) is used to describe a set of terms *i.e.* a subset of $T_{\Sigma \cup \mathcal{X}}$. If $N_{\mathcal{M}}$ denote the function giving the set of metavariable of a metaterm and T is a metaterm such that $N_{\mathcal{M}}(\mathsf{T}) \neq \varnothing$, we define:

$$[\![\mathsf{T}]\!] = \{x \in T_{\Sigma \cup \mathcal{X}} \mid N_{\mathcal{M}}(\mathsf{T}) = \{\overrightarrow{M}\} \wedge (\exists \overrightarrow{x} \in T^n_{\Sigma \cup \mathcal{X}} \wedge x = [\overrightarrow{M \mapsto x}]\mathsf{T})\}$$

This set is said to be the *meaning*[11] of the metaterm. In general, the brackets ($[\![\cdot]\!]$) are omitted and one must understand when reading a metaterm T that we are talking of the set $[\![T]\!]$. Therefore, the precise set of metavariable is often left implicit. For example, by abuse of notation we say:

$$c(M_1, M_2) = \{x \in T_{\Sigma \cup \mathcal{X}} \mid \exists \{x_1, x_2\} \subset T_{\Sigma \cup \mathcal{X}} \wedge x = c(x_1, x_2)\}$$

The set of terms with variables ($T_{\Sigma \cup \mathcal{X}}$) is denoted $T_\Sigma[\mathcal{X}]$. The variables are denoted using lowercase. Such a term represent a still unknown term. The term will be obtained by applying a substitution. Mathematically, such a term T containing n variables is a function from $T_\Sigma[\mathcal{X}]^n$ to $T_\Sigma[\mathcal{X}]$.

To define scopes of variables such that, for example, x in M does not clash with "another" x in a larger term (*e.g.* $M + x$), we define binders[12]. A binder b of arity n binds the n variables $x_1, ..., x_n$ within the term M when applied by $b(x_1, ..., x_n).M$. The arity of a binder is at least 1. In programming languages, examples of binders are the variable declaration instruction (*e.g.* `int v;`) with the scope going until the current bloc terminates or the definition of a function (*e.g.* `int f(int i, float f) {...}`) that binds several variables and whose scope is the body of the function.

[11]It is a semantics in the denotational sense defined later hence the use of the double brackets notation.
[12]there are other ways of introducing binding, see for example [3]

Definition 20, Term algebra with binders

Let $\mathcal{X}$ be a countable set of variables, Σ be the union of $\mathcal{C}$ the set of constructors and $\mathcal{B}$ the set of binders. A term algebra with binders $T_\Sigma[\mathcal{X}]$ is the smallest part of $(\Sigma \cup \mathcal{X})^*$ such that:

1. $\mathcal{X} \subset T_\Sigma[\mathcal{X}]$
2. $\{c \in \Sigma \mid ar(c) = 0\} \subset T_\Sigma[\mathcal{X}]$
3. $c \in \mathcal{C} \wedge ar(c) = n \geq 1 \wedge \overrightarrow{M} \in T_\Sigma[\mathcal{X}]^n \Rightarrow c(\overrightarrow{M}) \in T_\Sigma[\mathcal{X}]$
4. $b \in \mathcal{B} \wedge ar(b) = n \wedge \overrightarrow{x} \in \mathcal{X}^n \wedge M \in T_\Sigma[\mathcal{X}] \Rightarrow b(\overrightarrow{x}).M \in T_\Sigma[\mathcal{X}]$

In such a language, a term may contain two kind of variables: bound variables or free variables. A bound variable is a variable in the scope of a binder binding it. A free variable is a variable not in the scope of a binder binding it. The set of free variables is defined by induction as follows:

$$\begin{cases} \mathsf{FV}(x) = \{x\} & \text{if } x \in \mathcal{X} \\ \mathsf{FV}(c) = \varnothing & \text{if } c \in \{c \in \Sigma \mid ar(c) = 0\} \\ \mathsf{FV}(c(\overrightarrow{M})) = \bigcup_{i \in [\![1..n]\!]} \mathsf{FV}(M_i) & \text{if } c \in \{c \in \mathcal{C} \mid ar(c) = n \geq 1\} \\ \mathsf{FV}(b(\overrightarrow{x}).M) = \mathsf{FV}(M) \setminus \{\overrightarrow{x}\} & \text{if } b \in \{b \in \mathcal{B} \mid ar(b) = n\} \end{cases}$$

In a term, a bound name has no importance, it's only a link to its binder[13]. This is formalized by defining an equivalence relation $\equiv_\alpha$ on terms saying that two terms are equivalent if they only differ through bound names.

$$\begin{cases} x \equiv_\alpha T & \text{if } T = x & \wedge x \in \mathcal{X} \\ c \equiv_\alpha T & \text{if } T = c & \wedge c \in \{c \in \Sigma \mid ar(c) = 0\} \\ c(\overrightarrow{M}) \equiv_\alpha T & \text{if } T = c(\overrightarrow{N}) \wedge \forall i, M_i \equiv_\alpha N_i & \wedge c \in \{c \in \mathcal{C} \mid ar(c) = n \geq 1\} \\ b(\overrightarrow{x}).M \equiv_\alpha T & \text{if } T = b(\overrightarrow{y}).N \wedge [\overrightarrow{x \mapsto y}]M \equiv_\alpha N & \wedge b \in \{b \in \mathcal{B} \mid ar(b) = n\} \end{cases}$$

Then, we work on the corresponding quotient $T_\Sigma[\mathcal{X}]/\equiv_\alpha$[14]. It is not done in this course to simplify the presentation. But we sometimes refer to α-renaming to change some bound variables (using the notation $=_\alpha$).

Substitution definition must be must extended accordingly by:

4. $\sigma b(\overrightarrow{x}).M = b(\overrightarrow{x}).\sigma|_{\overrightarrow{x}}M$, for $b \in \mathcal{B}$, $ar(b) = n$, $\overrightarrow{x} \in \mathcal{X}^n$, $\sigma|_{\overrightarrow{x}}(y) = y$ if $y \in \{\overrightarrow{x}\}$ and $\sigma|_{\overrightarrow{x}}(y) = \sigma y$ otherwise

Notice that the substitution does not affect the variables bound by a binder. A substitution can add new bound variables, a phenomenon called capture. This happens when a variable x appears in the substitution result and is applied in the scope of a binder binding x. For example, if b is a binder of arity 1 and c is a constructor of arity 2:

$$[y \mapsto x]c(y, b(x).y) = c([y \mapsto x]y, [y \mapsto x]b(x).y) = c(x, b(x).[y \mapsto x]y) = c(x, b(x).x)$$

Even if they are both created by the same substitution, the second occurrence of x is very different from the first as it is bound whereas the first is free. In general, we try to avoid capture by renaming the variable bound in the term before the substitution.

$$\begin{aligned} [y \mapsto x]c(y, b(x).y) &= c([y \mapsto x]y, [y \mapsto x]b(x).y) =_\alpha c(x, [y \mapsto x][x \mapsto z]b(x).y) \\ &= c(x, [y \mapsto x]b(z).y) = c(x, b(z).x) \end{aligned}$$

[13]There exists notations without names, see for example [2].

[14]It is the set of equivalent classes of the equivalence relation.

6 Syntax

Often, we distinguish between the concrete syntax and the abstract syntax of a language. The concrete syntax describes precisely the strings of the language in terms of symbols (the subset of the free monoid) whereas an abstract syntax describes abstractly the possible constructs on the form of an term algebra (see below). The concrete syntax describes how to write a string of the language while the abstract syntax describes the essence of this string. For example, the C string "`if (a) then { 1 } else { 0 }`" can be represented by the abstract syntax "$\mathsf{if}(a, 1, 0)$".

Concrete syntax focuses on the interaction with the (human) user of the syntax whereas abstract is aimed at the computational treatments.

6.1 Concrete syntax

For complex languages, such usual programming languages, the syntax is separated in two levels. In a first level, we have raw symbols that form tokens (sometimes called syntactic units). And in a second level, these tokens are combined to form statements. For example, the first level would use the alphabet $\{\mathtt{a}, \mathtt{b}, ..., \mathtt{z}\}$ whereas the second could be $\{\mathtt{if}, \mathtt{then}, \mathtt{else}\}$.

Furthermore, the first level is often built as a collection of languages. For example, we could build a simple expression language $\mathcal{L}_{exp}$ as the union of $\mathcal{L}_{int} = \{x \in \{\mathtt{0}, \mathtt{1}, \mathtt{2}, ..., \mathtt{9}\}^* \mid |x| \geq 1\}$, $\mathcal{L}_{fixed} = \{x\,\mathtt{.}\,y \mid x \in \mathcal{L}_{int} \wedge y \in \mathcal{L}_{int}\}$, $\mathcal{L}_{op} = \{\mathtt{+}, \mathtt{-}, \mathtt{*}, \mathtt{/}\}$ and $\mathcal{L}_{par} = \{\mathtt{(}, \mathtt{)}\}$.

More formally, the scanner is a small generalization of a FA called a *transducer*. A transducer is a FA with an output alphabet and where transitions can also produce an output values. The input alphabet will generally be composed of characters and the output alphabet will be composed of tokens.

The concrete syntax focusses on interaction with the (human) user. As such it must be readable and efficient for its user. It must define precisely how to understand ambiguous strings. For example, the string `1+2*5` can be understood as (`1+2`)`*5` or `1+`(`2*5`).

Consult `http://www.infoq.com/presentations/Language-Design`.

[TODO: Transducer, better link with FA]

6.2 Abstract syntax

The abstract syntax must define the *essential* content of a string. It is mainly used to understand and manipulate a string.

[TODO: link with term algebra]

7 Giving meanings to syntactic objects

There is mainly three ways of defining the semantics of a term:

1. axiomatic semantics: some logical assertions express properties of terms. Such a semantics is defined by systems of equations describing the effect of each syntactic construction to logical assertions. The most well-known is Hoare triple[15] where $\{Pre\}T\{Post\}$ expresses that if

[15]Consult `http://en.wikipedia.org/wiki/Hoare_logic`.

Pre is true before the execution of T and T terminates then *Post* is true after its execution. This kind of semantics gives a macroscopic vision of the meaning (generally partial) and is mainly used to study properties like *consistency*, *completeness*, *compositionality*, ...

2. denotational semantics[16]: each term is mapped to an object of a known (mathematical) space (*e.g.* sets, universal algebra, domain, category, ...). The semantics is given by a projection called an interpretation and denoted $[\![.]\!]$ or $\mathcal{I}(.)$. This gives an abstract vision of the meaning, we do not really know of the computation happens, we only have an image of it. It is mainly used to study meta-theory such as the equivalence of terms, fixed-point theory, ... A denotational semantics is not easy to construct as it generally requires *compositionality* (the meaning of a term is the composition of the meaning of its subterms).

3. operational semantics[17]: how computation behaves (the sequence of states). Often it is defined by computation rules (transition system) where each term either reduces to another (smaller) term or is a value. There exists two kinds of operational semantics: *small-step* or *big-step*. In the former every step of computation is described whereas the latter specify directly the result for each term. Both gives a microscopic vision of the meaning useful to study termination, non-determinism, cost of computation...

`http://en.wikipedia.org/wiki/Semantics_(computer_science)`

8 Transition systems

Definition 21, Transition systems

A transition system[a] is a pair $(S, \to)$ of a set S (of states) and a binary relation $\to$ of S ($\to \subset S \times S$).

A pair (p, q) of $\to$ is noted infix $p \to q$ and we speak of a transition from state p to state q.

[a]Also known as reduction system.

In this course, states will be terms.

A transition system is a directed graph where vertices are the states and arcs are defined by the transition relation. If we add *initial* and *final* states, we fall back on automaton when the set of state and of transition are finite. Therefore, a transition system is deterministic iff for all state s there exists at most one state s' such that $s \to s'$.

A term T is irreducible or normal, if there exists no term it can reduce to ($T \not\to$). If a term T reduces to another term T' which is normal, T' is called a normal form of T. A term is normalizable (resp. strongly normalizable) if it has at least one normal form (resp. it has only finite reduction sequences). A term may have no normal form (it has only infinite reductions). A term may have both normal forms and infinite reductions (it is normalizable but not strongly normalizable).

A reduction sequence (or trace) is a sequence of terms T_1, ..., T_n such that $T_1 \to T_2$, ..., $T_{n-1} \to T_n$. It is sometimes denoted $T_1 \to^n T_n$ when we do not care about the intermediary terms. The reflexive and transitive closure, denoted $\to^*$ is the smallest relation including $\to$ being reflexive and transitive. It contains all the reduction sequences.

[16]Consult `http://en.wikipedia.org/wiki/Denotational_semantics`.
[17]Consult `http://en.wikipedia.org/wiki/Operational_semantics`.

$$\begin{tikzcd}
 & T \arrow[dl, dotted] \arrow[dr, dotted] & \\
T_1 \arrow[dr, dotted] & & T_2 \arrow[dl, dotted] \\
 & T' &
\end{tikzcd}$$

Figure 4: Confluence

Definition 22, Strongly / weakly normalizing

A transition system is strongly normalizing if all its reduction sequences are finite. It is weakly normalizing if all terms have a normal form.

Notice that strongly normalizing implies weakly normalizing.

A transition system is confluent in T iff when $T \to^* T_1$ and $T \to^* T_2$, there exists a T' such that $T_1 \to^* T'$ and $T_2 \to^* T'$. It is confluent if it is confluent on each term. This is often represented by the schema in figure 4. Notice that confluence implies uniqueness of normal forms. Confluent reductions are interesting because whatever the path of reduction you chose you will end up with the same result.

9 Inference rule and proof

Definition 23, Judgement

A judgment is a logical assertion.

For example, $\mathit{Term} \to \mathit{OtherTerm}$ is a (reduction) judgement.

Definition 24, Inference rule

An inference rule is a collection of $n+1$ judgments J_1, ..., , J_n and J such that $J_1 \wedge \ldots \wedge J_n \Rightarrow J$. The left hand judgements J_1, ..., , J_n are called the premises whereas the right hand J is called the conclusion. An inference rule with no premise is called an axiom. An inference rule is written:

$$\frac{J_1 \qquad \cdots \qquad J_n}{J}$$

Consult `http://en.wikipedia.org/wiki/Inference_rule`.

Definition 25, Proof

A derivation (or proof) is a tree of inference rules where the leaves are axioms.

For example:

$$\dfrac{\dfrac{}{J_1} \qquad \dfrac{}{J_2} \qquad \cdots \qquad \dfrac{\dfrac{}{J_3} \qquad \cdots \qquad \dfrac{}{J_4}}{J_5}}{J_6}$$

Such a tree can be built from axioms to conclusion following a method called forward chaining. In this case, we are doing a blind search in the judgment space as we do not know how to proceed to the conclusion. Another method proceed from conclusion to axioms and is called backward chaining. This time the search is guided at each step by the form of the current conclusion.

Glossary

A

Abstract syntax 19, 22

Accepted 5, 13, 22

Accepting 5, 22

Accessible 5, 22

Alphabet A set whose elements (the symbols) are used to build strings. It provides the building blocks to define formal languages. 2–4, 9, 13, 19, 22, 25

Ambiguous 11, 22

Arity 15, 22

Axiom 10, 21, 22

Axiomatic semantics 19, 22

B

Backward chaining 22

Binder 17, 22

C

Capture 18, 22

Chomsky hierarchy 10, 22

Closed term 16, 22

Complete 6, 22

Conclusion 21, 22

Concrete syntax 19, 22

Configuration 13, 22

Confluent 21, 22

Constant 15, 22

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison Wesley, August 2006.

[2] N. Bourbaki. *Théorie des ensembles.* Eléments de mathématique. Springer, 2008.

[3] Robert Harper. *Practical Foundations for Programming Languages.* Cambridge University Press, New York, NY, USA, 2012.

[4] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2007.