1 Recursive automata

A finite automaton can be seen as a program with only a finite amount of memory. A recursive automaton is like a program which can use *recursion* (calling procedures recursively), but again over a finite amount of memory in its variable space. Note that the recursion, which is typically handled by using a stack, gives a limited form of *infinite* memory to the machine, which it can use to accept certain non-regular languages. It turns out that the recursive definition of a language defined using a context-free grammar precisely corresponds to recursion in a finite-state recursive automaton.

A recursive automaton over Σ is made up of a finite set of NFAs that can call each other (like in a programming language), perhaps recursively, in order to check if a word belongs to a language.

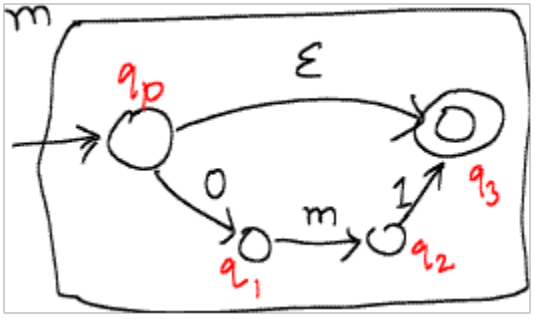
Formally, a recursive automaton over Σ is a tuple $(M, main, \{(Q_m, \Sigma \cup M, \delta_m, q_0^m, F_m)\}_{m \in M})$, where

- M is a finite set of module names,
- $main \in M$ is the initial module,
- For each $m \in M$, $A_m = (Q_m, \Sigma \cup M, \delta_m, q_0^m, F_m)$ is an NFA over the alphabet $\Sigma \cup M$.
- For any $m, m' \in M$, $m \neq m'$, $Q_m \cap Q_{m'} = \emptyset$ (the set of states of different modules are disjoint).

Intuitively, we view a recursive automaton as a set of procedures/modules, where the execution starts with the *main*-module, and the automaton processes the word by calling modules recursively.

Example

Let $\Sigma = \{a, b\}$ and let $L = \{0^n 1^n \mid n \in \mathbb{N}\}$. L is accepted by the following recursive automaton.



Why? The recursive automaton consists of single module, which is also the *main* module. The module either accepts ϵ, or reads 0, calls itself, and after returning from the call, reads 1 and reaches a final state (at which point it can return if it was called). In order to accept, we require the run to return from all calls and reach the final state of module *main*.

For example, the recursive automaton accepts 01 because of the following run: $q_0 - 0 - > q_1 - (call\ m) - > q_0 - \epsilon - > q_3 - (return) - > q_2 - 1 - > q_3$. Note that the transition $q_1 - m - > q_3$ calls module m; module m starts with its initial state, will process letters (perhaps recursively calling more modules), and when it reaches a final state will return to the state q_3 in the calling module.

Formal definition of acceptance

Formally, let $A = (M, main, \{(Q_m, \Sigma \cup M, \delta_m, q_0^m, F_m)\}_{m \in M})$ be a recursive automaton.

We define a run of A on a word w. Since the modules can call each other recursively, we define the run using a stack. When A is in state q and calls a module m using the transition $q - m - > q'$, we push q' onto the stack so that we know where to go to when we return from the call. When we return from a call, we pop the stack and go to the state stored on the top of the stack.

Formally, let $Q = \bigcup_{m \in M} Q_m$ be the set of all states in the automaton. A stack is a word $s \in Q^*$. A configuration is a pair (q, s) where $q \in Q$ and s is a stack.

We say that a word w is accepted by A provided we can write $w = y_1 \dots y_k$, such that each $y_i \in \Sigma \cup \{\epsilon\}$, and there is a sequence of $k+1$ configurations $(q_0, s_0), \dots (q_k, s_k)$ such that

- $q_0 = q_0^{main}$ and $s_0 = \epsilon$
  We start with the initial state of the main module with stack being empty.
- $q_k \in F_{main}$ and $s_k = \epsilon$
  We end with a final state of the main module with stack being empty (i.e. we expect all calls to have returned).
- For every $i < k$, one of the following must hold:
  - $q_i \in Q_m$, $\delta_m(q_i, y_{i+1}) = q_{i+1}$, and $s_{i+1} = s_i$.
  - $q_i \in Q_m$, $y_{i+1} = \epsilon$, $\delta_m(q_i, m') = q'$, $q_{i+1} = q_0^{m'}$ and $s_{i+1} = q'.s_i$.
  - $q_i \in F_m$, $y_{i+1} = \epsilon$ and $s_{i+1} = q_{i+1}.s_i$.

2 CFGs and recursive automata

We will now show that context-free grammars and recursive automata accept precisely the same class of languages.

2.1 From CFGs to recursive automata

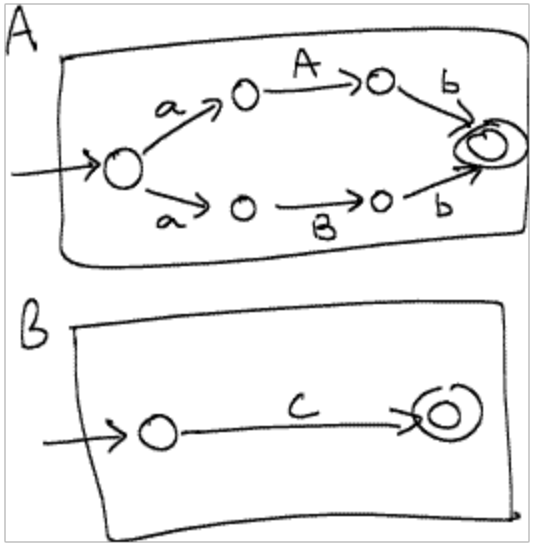
Given a CFG, we want to construct a recursive automaton for the language generated by the CFG. Let us first do this for an example.

Consider the grammar (where A is the start variable) which generates $\{a^n c b^n \mid n \in \mathbb{N}\}$:

$$\begin{aligned} A &\rightarrow aAb \mid aBb \\ B &\rightarrow c \end{aligned}$$

Each variable in the CFG corresponds to a language; this language is recursively defined using other variables. We hence look upon each variable as a module; and define modules that accept words by calling other modules recursively.

For example, the recursive automaton for the above grammar is:



Formally, let $G = (V, \Sigma, R, S)$ be a context free grammar. Let $A_G = (M, main, \{(Q_m, \Sigma \cup M, \delta_m, q_0^m, F_m)\}_{m \in M})$ where $M = V$, $main = S$ and for each $X \in M = V$, let $\langle Q_m, \Sigma \cup M, \delta_m, q_0^m, F_m)\rangle$ be an NFA that accepts the (finite, and hence) regular language $L_X = \{w | (X \to w) \in R\}$.

Then $L(G) = L(A_G)$.

2.2 From recursive automata to CFGs

Let $A = (M, main, \{(Q_m, \Sigma \cup M, \delta_m, q_0^m, F_m)\}_{m \in M})$ be a recursive automaton. We construct a CFG $G_A = (V, \Sigma, R, S)$ with $V = \{X_q \mid q \in \bigcup_{m \in M} Q_m\}$.

Intuitively, the variable X_q will represent the set of all words accepted by starting in state q and ending in a final state of the module q is in (however, on recursive calls to this module, we still enter at the original initial state of the module).

R is the following set of rules:

- If $q' \in \delta_m(q, a)$, then the rule $X_q \to aX_{q'}$ is in R, for any $m \in M$, $q, q' \in Q_m$, $a \in \Sigma \cup \{\epsilon\}$.

  Intuitively, a transition within a module is simulated by generating the letter on the transition and generating a variable that stands for the language generated from the next state.

- If $q' \in \delta_m(q, m')$, then the rule $X_q \to X_{q_0^{m'}} X_{q'}$ is in R, for any $m, m' \in M$, $q, q' \in Q_m$.

  Intuitively, if $q' \in \delta_m(q, m')$, then X_q can generate a word of the form xy where x is accepted using a call to module m and y is accepted from the state q'.

- For any $q \in \bigcup_{m \in M} F_m$, $X_q \to \epsilon$ belongs to R.

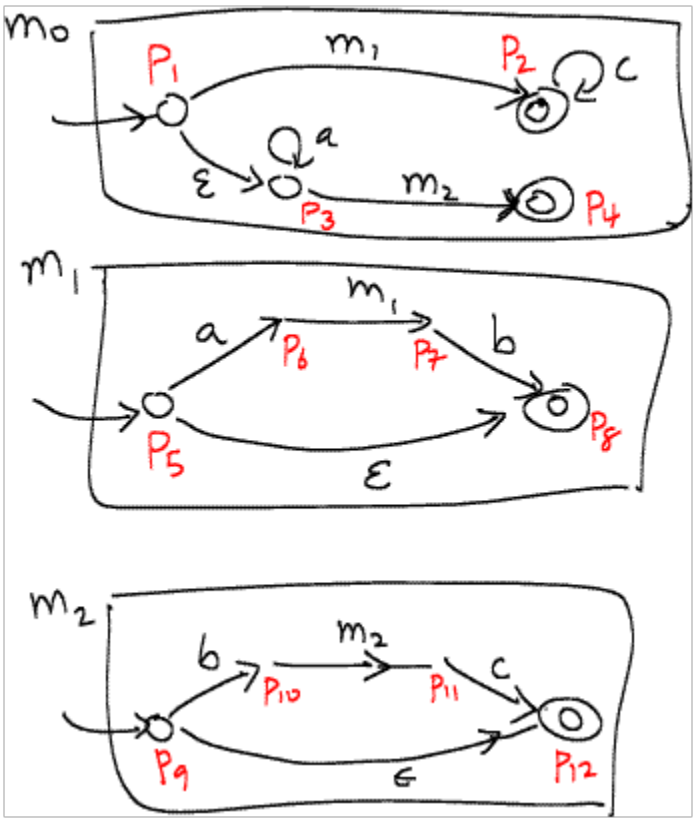
  When at a final state, we can stop generating letters.

The initial variable S is $X_{q_0^{main}}$.

Then $L(G_A) = L(A)$.

An example conversion from PDAs to CFGs

Consider the following recursive automaton, which accepts the language $\{a^i b^j c^k \mid i = j \text{ or } j = k\}$.



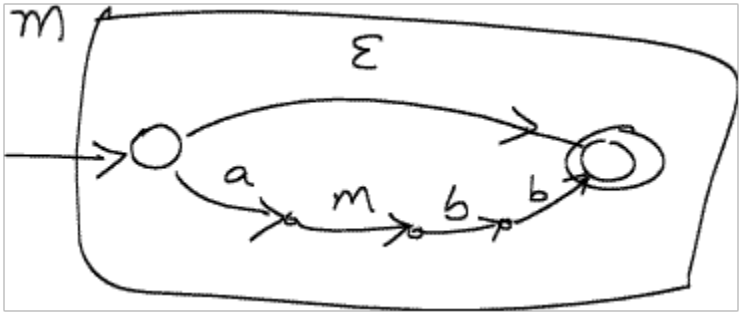
The grammar corresponding to it is:

$$
\begin{aligned}
X_{p1} &\rightarrow X_{p5}X_{p2} \mid X_{p3} \\
X_{p2} &\rightarrow cX_{p2} \mid \epsilon \\
X_{p3} &\rightarrow aX_{p3} \mid X_{p9}X_{p4} \\
X_{p4} &\rightarrow \epsilon \\
X_{p5} &\rightarrow aX_{p6} \mid X_{p8} \\
X_{p6} &\rightarrow X_{p5}X_{p7} \\
X_{p7} &\rightarrow bX_{p8} \\
X_{p8} &\rightarrow \epsilon \\
X_{p9} &\rightarrow bX_{p10} \mid X_{p12} \\
X_{p10} &\rightarrow X_{p9}X_{p11} \\
X_{p11} &\rightarrow cX_{p12} \\
X_{p12} &\rightarrow \epsilon
\end{aligned}
$$

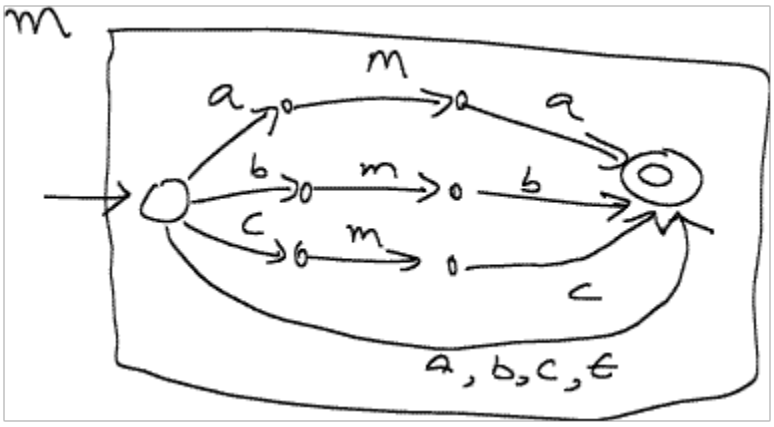
where the start variable is X_{p1}.

3 More examples

Example 1: Let us design a recursive automaton for the language $L = \{a^n b^{2n} \mid n \in \mathbb{N}\}$. We would like to generate this recursively. How do we generate $a^{n+1}b^{2n+2}$ using a procedure to generate $a^n b^{2n}$? We read a followed by a call to generate $a^n b^{2n}$, and follow that by generating two b's. The "base-case" of this recursion is when $n = 0$, when we must accept ϵ. This leads us to the following automaton:



Example 2: Let us design a recursive automaton for the language $L = \{w \in \{a, b, c\}^* \mid$ *w is a palindrome* $\}$. Thinking recursively, the smallest palindromes are ϵ, a, b, c, and we can construct a longer palindrome by generating awa, bwb, cwc, where w is a smaller palindrome. This give us the following recursive automaton:



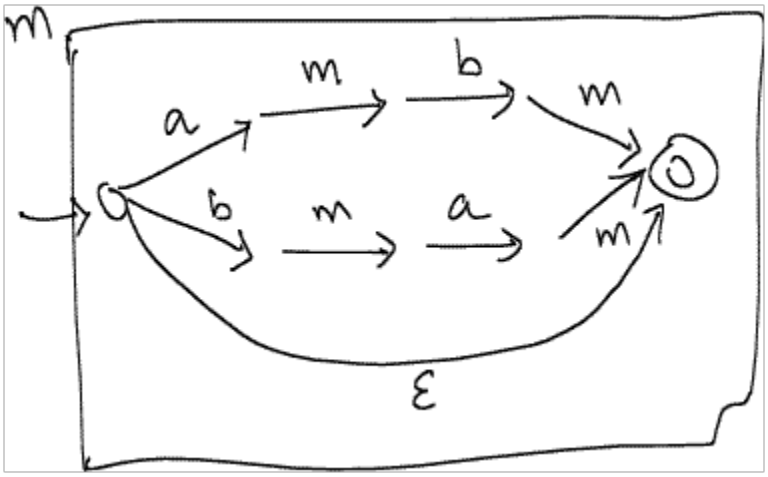
Example 3: Let us design a recursive automaton for the language L containing all strings $w \in \{a, b\}^*$ that has an equal number of a's and b's.

Let w be a string, of length at least one, with equal number of a's and b's.

Case 1: w starts with a. As we read longer and longer prefixes of w, we have the number of a's seen is more than the number of b's seen. This situation can continue, but we must reach a place when the number of a's seen is precisely the number of b's seen (at worst at the end of the word). Let us consider some prefix longer than a where this happens. Then we have that $w = aw_1bw_2$, where the number of a's and b's in aw_1b is the same, i.e. the number of a's and b's in w_1 are the same. Hence the number of a's and b's in w_2 are also the same.

Case 2: If w starts with b, then by a similar argument as above, $w = bw_1aw_2$ for some (smaller) words w_1 and w_2 in L.

Hence any word in L of length at least one is of the form aw_1bw_2 or bw_1aw_2, where $w_1, w_2 \in L$, and are smaller. Also, note ϵ is in L. So this gives us the following recursive automaton:



4 Recursive automata and pushdown automata

The definition of acceptance of a word by a recursive automaton employs a *stack*, where the target state gets pushed on a call-transition, and gets popped when the called module returns. An alternate way (and classical) way of defining automata models for context-free languages directly uses a stack. A *pushdown automaton* is a non-deterministic automaton with a finite set of control states, and where transitions are allowed to push and pop letters from a finite alphabet Γ (Γ is fixed, of course) onto the stack. It should be clear that a recursive automaton can be easily simulated by a pushdown automaton (we simply take the union of all states of the recursive automaton, and replace call transitions $q - m- > q'$ with an explicit push-transition that pushes q' onto the stack and explicit pop transitions from the final states in F_m to q' on popping q'.

It turns out that pushdown automata can be converted to recursive automata (and hence to CFGs) as well. This is a fact worth knowing! But we will not define pushdown automata formally, nor show this direction of the proof.