Lecture 16: Recursive automatas

17 March 2009

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.

1.1 Formal definition of RAs

A recursive automaton (RA) 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.

Definition 1.1 A ***recursive automaton*** (RA) over Σ is a tuple

$$\left(M, \mathbf{main}, \left\{ \mathtt{M}_M \;\middle|\; m \in M \right\}\right),$$

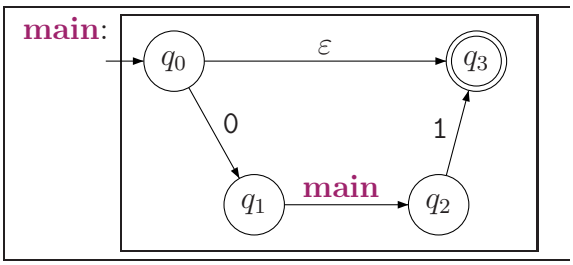
where

- M is a finite set of module names,
- $\mathbf{main} \in M$ is the initial module,
- For each $m \in M$, there is an associated automaton $\mathtt{M}_m = (Q_m, \Sigma \cup M, \delta_m, q_0^m, F_m)$ which is an NFA over the alphabet $\Sigma \cup M$. In other words, Q_m is a finite set of states, $q_0^m \in Q_m$ is the initial state (of the module m), $F_m \subseteq Q_m$ is the set of final states of the module m (from where the module can return), and $\delta_m : Q_m \times (\Sigma \cup M \cup \{\epsilon\}) \to 2^{Q_m}$ is the (non-deterministic) transition function.
- For any $m, m' \in M$, $m \neq m'$ we have $Q_m \cap Q_{m'} = \emptyset$ (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.

1.1.1 Example of a recursive automata

Let $\Sigma = \{0, 1\}$ and let $L = \left\{ 0^n 1^n \;\middle|\; n \in \mathbb{N} \right\}$. The language 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 the module **main**.

For example, the recursive automaton accepts 01 because of the following execution

$$q_0 \xrightarrow{0} q_1 \xrightarrow{\text{call main}} q_0 \xrightarrow{\epsilon} q_3 \xrightarrow{\text{return}} q_2 \xrightarrow{1} q_3.$$

Note that using a transition of the form $q \xrightarrow{\mathtt{m}} q'$ calls the module m; the 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' in the calling module.

1.1.2 Formal definition of acceptance

Stack. We first need the concept of a ***stack***. A stack s is a list of elements. The ***top of the stack*** (TOS) is the first element in the list, denoted by topOfStack(() s). Pushing an element x into a stack (i.e., ***push*** operation) s, is equivalent to creating a new list, with the first element being x, and the rest of the list being s. We denote the resulting stack by push(s, x). Similarly, popping the stack s (i.e., ***pop*** operation) is the list created from removing the first element of s. We will denote the resulting stack by pop(s).

We denote the empty stack by $\langle\rangle$. A stack containing the elements x, y, z (in this order) is written as $s = \langle x, y, z\rangle$. Here topOfStack$(s) = x$, pop$(s) = \langle y, z\rangle$ and push$(s, b) = \langle b, x, y, z\rangle$.

Acceptance. Formally, let $\mathtt{D} = \left(M, \textbf{main}, \left\{ (Q_m, \Sigma \cup M, \delta_m, q_0^m, F_m) \;\middle|\; m \in M \right\}\right)$ be a recursive automaton.

We define a run of D on a word w. Since the modules can call each other recursively, we define the run using a stack. When D is in state q and calls a module m using the transition $q \stackrel{*}{\Longrightarrow} mq'$, 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 D. A ***configuration*** of D is a pair (q, s) where $q \in Q$ and s is a stack.

We say that a word w is ***accepted*** by D 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

- $q_0 = q_0^{\textbf{main}}$ and $s_0 = \langle\rangle$.

  We start with the initial state of the main module with the stack being empty.

- $q_k \in F_{\textbf{main}}$ and $s_k = \langle\rangle$.

  We end with a final state of the main module with the stack being empty (i.e. we expect all calls to have returned).

- For every $i < k$, one of the following must hold:

  Internal: $q_i \in Q_m$, $q_{i+1} \in \delta_m(q_i, y_{i+1})$, and $s_{i+1} = s_i$.

  Call: $q_i \in Q_m$, $y_{i+1} = \epsilon$, $q' \in \delta_m(q_i, m')$, $q_{i+1} = q_0^{m'}$ and $s_{i+1} = \text{push}(s_i, q')$.

  Return: $q_i \in F_m$, $y_{i+1} = \epsilon$, $q_{i+1} = \text{topOfStack}(s_i)$ and $s_{i+1} = \text{pop}(s_i)$.

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 Converting a CFG into a recursive automata

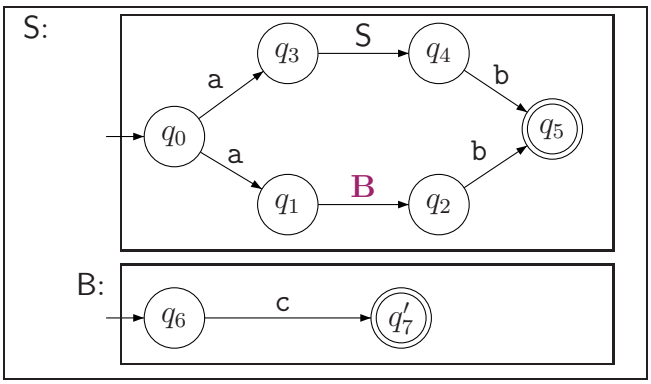
Given a CFG, we want to construct a recursive automaton for the language generated by the CFG. Let us first do this for an example.

Consider the grammar (where S is the start variable) which generates $\left\{ \mathtt{a}^n\mathtt{c}\mathtt{b}^n \;\middle|\; n \in \mathbb{N} \right\}$:

$$\begin{aligned} \implies \mathsf{S} &\to \mathtt{a}\mathsf{S}\mathtt{b} \mid \mathtt{a}\mathsf{B}\mathtt{b} \\ \mathsf{B} &\to \mathtt{c}. \end{aligned}$$

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:



(Here S is the main modules of the recursive automaton.)

Formal construction. Let $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})$ be the given context free grammar.

Let $\mathtt{M}_{\mathcal{G}} = \left(M, \mathsf{S}, \left\{(Q_m, \Sigma \cup M, \delta_m, q_0^m, F_m) \;\middle|\; m \in M\right\}\right)$ where $M = \mathcal{V}$, and the main module is S. Furthermore, for each $\mathsf{X} \in M$, let $\mathtt{M}_{\mathsf{X}} = \left(Q_{\mathsf{X}}, \Sigma \cup M, \delta_{\mathsf{X}}, q_0^{\mathsf{X}}, F_{\mathsf{X}}\right)$ be an NFA that accepts the (finite, and hence) regular language $L_{\mathsf{X}} = \left\{w \;\middle|\; (\mathsf{X} \to w) \in \mathcal{R}\right\}$.

Let us elaborate on the construction of $\mathtt{M}_{\mathsf{X}}$. We create two special states $q_{\text{init}}^{\mathsf{X}}$ and $q_{\text{final}}^{\mathsf{X}}$. Here $q_{\text{init}}^{\mathsf{X}}$ is the initial state of $\mathtt{M}_{\mathsf{X}}$ and $q_{\text{final}}^{\mathsf{X}}$ is the accepting state of $\mathtt{M}_{\mathsf{X}}$. Now, consider a rule $(\mathsf{X} \to w) \in \mathcal{R}$. We will introduce a path of length $|w|$ in $\mathtt{M}_{\mathsf{X}}$ (corresponding to w) leading from $q_{\text{init}}^{\mathsf{X}}$ to $q_{\text{final}}^{\mathsf{X}}$. Creating this path requires introducing new "dummy" states in the middle of the path, if $|w| > 1$. The ith transition along this path reads the ith character of w. Naturally, if this ith character is a variable, then this edge would correspond to a recursive call to the corresponding module. As such, if the variable X has k rules in the grammar $\mathcal{G}$, then $\mathtt{M}_{\mathsf{X}}$ would contain k disjoint paths from $q_{\text{init}}^{\mathsf{X}}$ to $q_{\text{final}}^{\mathsf{X}}$, corresponding to each such rule. For example, if we have the derivation $(\mathsf{X} \to \epsilon) \in \mathcal{R}$, then we have an ϵ-transition from $q_{\text{init}}^{\mathsf{X}}$ to $q_{\text{final}}^{\mathsf{X}}$.

2.2 Converting a recursive automata into a CFG

Let $\mathtt{D} = (M, \textbf{main}, \{(Q_m, \Sigma \cup M, \delta_m, q_{\text{init}}^m, F_m)\}_{m \in M})$ be a recursive automaton. We construct a CFG $\mathcal{G}_{\mathtt{D}} = (\mathcal{V}, \Sigma, \mathcal{R}, \mathsf{S})$ with $\mathcal{V} = \left\{\mathsf{X}_q \mid q \in \bigcup_{m \in M} Q_m\right\}$.

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).

The set of rules R is generated as follows.

- **Regular transitions.** For any $m \in M$, $q, q' \in Q_m$, $c \in \Sigma \cup \{\epsilon\}$, if $q' \in \delta_m(q, c)$, then the rule $\mathsf{X}_q \to c\mathsf{X}_{q'}$ is added to $\mathcal{R}$.

  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.

- **Recursive call transitions.** for all $m, m' \in M$ and $q, q' \in Q_m$, if $q' \in \delta_m(q, m')$, then the rule $\mathsf{X}_q \to \mathsf{X}_{q_{\text{init}}^{m'}} \mathsf{X}_{q'}$ is in R,

  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'.

- **Acceptance/return rules.**

  For any $q \in \bigcup_{m \in M} F_m$, we add $\mathsf{X}_q \to \epsilon$ to $\mathcal{R}$.

  When arriving at a final state, we can stop generating letters and return from the recursive call.

The initial variable S is $\mathsf{X}_{q_{\text{init}}^{\textbf{main}}}$; that is, the variable corresponding to the initial state of the main module.

We have a CFG and it is not too hard to see intuitively that the language generated by this grammar is equal to the RA D language. We will not prove it formally here, but we state the result for the sake of completeness.

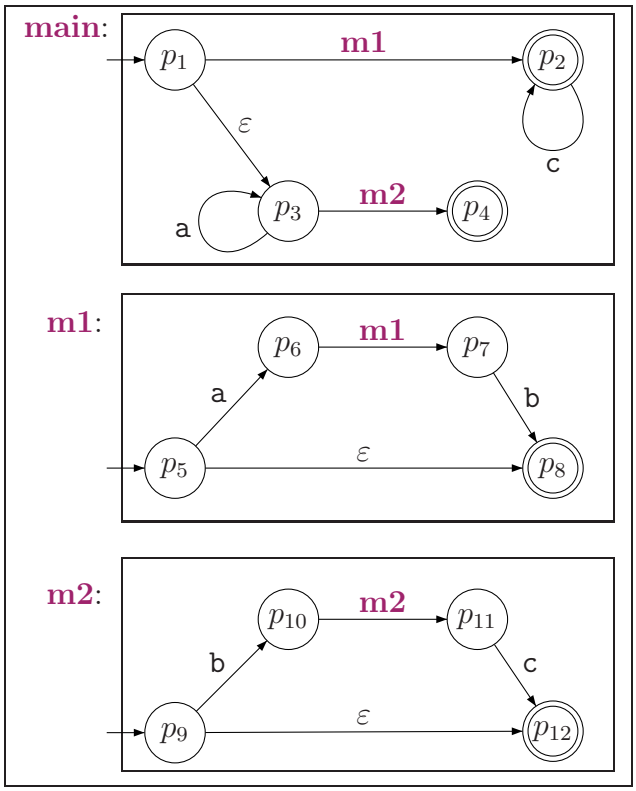
Lemma 2.1 $\mathsf{L}(\mathcal{G}_{\mathtt{D}}) = \mathsf{L}(\mathtt{D})$.

2.2.1 An example of conversion of a RA into a CFG

Consider the following recursive automaton, which accepts the language

$$\left\{ \mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \;\middle|\; i = j \text{ or } j = k \right\},$$

and the grammar generating it.



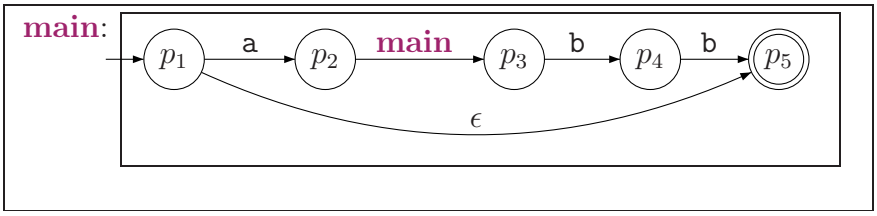
$$\begin{aligned}
\mathsf{X}_{p1} &\to \mathsf{X}_{p5}\mathsf{X}_{p2} \mid \mathsf{X}_{p3} \\
\mathsf{X}_{p2} &\to \mathtt{c}\mathsf{X}_{p2} \mid \epsilon \\
\mathsf{X}_{p3} &\to \mathtt{a}X_{p3} \mid \mathsf{X}_{p9}X_{p4} \\
\mathsf{X}_{p4} &\to \epsilon \\
\mathsf{X}_{p5} &\to \mathtt{a}\mathsf{X}_{p6} \mid \mathsf{X}_{p8} \\
\mathsf{X}_{p6} &\to \mathsf{X}_{p5}\mathsf{X}_{p7} \\
\mathsf{X}_{p7} &\to \mathtt{b}X_{p8} \\
\mathsf{X}_{p8} &\to \epsilon \\
\mathsf{X}_{p9} &\to \mathtt{b}\mathsf{X}_{p10} \mid \mathsf{X}_{p12} \\
\mathsf{X}_{p10} &\to \mathsf{X}_{p9}\mathsf{X}_{p11} \\
\mathsf{X}_{p11} &\to \mathtt{c}X_{p12} \\
\mathsf{X}_{p12} &\to \epsilon
\end{aligned}$$

The start variable is X_{p1}.

3 More examples

3.1 Example 1: RA for the language $\mathtt{a}^n\mathtt{b}^{2n}$

Let us design a recursive automaton for the language $L = \left\{ \mathtt{a}^n\mathtt{b}^{2n} \;\middle|\; n \in \mathbb{N} \right\}$. We would like to generate this recursively. How do we generate $\mathtt{a}^{n+1}\mathtt{b}^{2n+2}$ using a procedure to generate $\mathtt{a}^n\mathtt{b}^{2n}$? We read a followed by a call to generate $\mathtt{a}^n\mathtt{b}^{2n}$, 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:

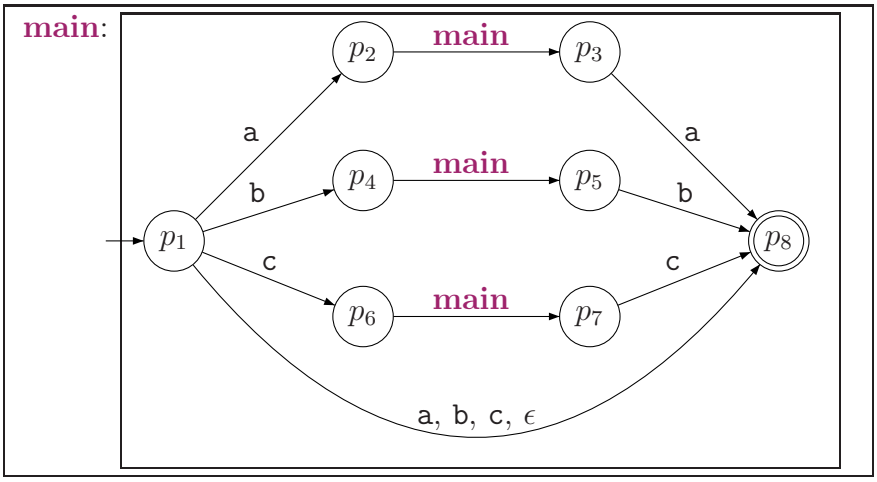


3.2 Example 2: Palindrome

Let us design a recursive automaton for the language

$$L = \left\{ w \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}^* \;\middle|\; w \text{ is a palindrome} \right\}.$$

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:



3.3 Example 3: $\#_a = \#_b$

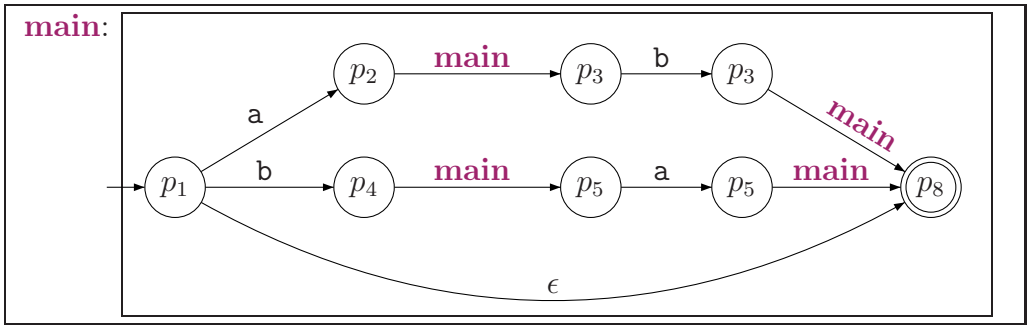
Let us design a recursive automaton for the language L containing all strings $w \in \{\mathtt{a}, \mathtt{b}\}^*$ that has an equal number of a's and b's.

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 $w = \mathtt{a}w_1\mathtt{b}w_2$, where the number of a's and b's in $\mathtt{a}w_1\mathtt{b}$ 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, $w = \mathtt{b}w_1\mathtt{a}w_2$ for some (smaller) words w_1 and w_2 in L.

Hence any word w in L of length at least one is of the form $\mathtt{a}w_1\mathtt{b}w_2$ or $\mathtt{b}w_1\mathtt{a}w_2$, where $w_1, w_2 \in L$, and they are strictly shorter than w. 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*** (PDA) 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 $q \xrightarrow{\mathtt{m}} q'$ 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.