1 Introduction

The collection of regular languages, while interesting, is not rich enough to be useful. For instance, even a simple language such as $\{a^n b^n \mid n \geq 0\}$ is not regular (though, to be frank, we have no way of proving this – yet). The notion of a *grammar* and in particular of a *context free grammar* provides a more general way of recursively defining languages which is still "mechanical" in some sense. In this lecture we explore this concept.

2 Rules

A *rule* or *production rule* is an expression of the form:

$$A \to w.$$

The left hand side, A must be an element of a finite set V of *variables* or *nonterminals* (generally written as upper case letters). The right hand side w must be a string in $(V \cup \Sigma)^*$ where Σ is some fixed finite alphabet disjoint from V (whose elements are generally written as lower case letters). The elements of Σ are also called *terminals*. For instance, all of the following are rules (over appropriate alphabets):

$$S \to aSb,\ S \to \lambda,\ T \to aTbT,\ T \to aPbQcR.$$

A rule can be *applied* to any string in $(V \cup \Sigma)^*$ which contains the symbol occurring on the left hand side of the rule. The result of such an application is to replace *any one* instance of the left hand side by the string on the right hand side. That is, if the rule $A \to w$ is applied to the string uAv, the result is the string uwv. We write:

$$uAv \overset{A \to w}{\Longrightarrow} uwv$$

or just $uAv \Rightarrow uwv$ if we don't care to mention the rule that was applied.

3 Grammars

A *context free grammar*, G, is just some set of rules over fixed sets of nonterminal and terminal symbols, together with a single distinguished nonterminal symbol (universally denoted S) called the *start* symbol.

A *derivation* in (or of G) is a sequence of rule applications:

$$v = v_1 \Rightarrow v_2 \Rightarrow v_3 \Rightarrow \cdots \Rightarrow v_n = w$$

where each rule is one of the rules of G which we write in shorthand as $v \overset{*}{\Rightarrow} w$, and in this case we say that w is *derivable* from v (in G).

Formally we could define derivability recursively: v is derivable from itself (base), and if $u = xAy$ is derivable from v and $A \to w$ is a rule of G then xwy is derivable from v.

The *language of* G, $L(G)$ is:

$$\{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$$

i.e. the set of strings over the terminal symbols that are derivable from the start symbol.

4 Examples

The language of:

$$S \to aS,\ S \to bS,\ S \to \lambda$$

is $(\boldsymbol{a} \cup \boldsymbol{b})^*$.

When a grammar has a number of rules with the same LHS they are frequently collected together, e.g. the preceding example could be written:

$$S \to aS \mid bS \mid \lambda$$

The language of

$$S \to aS \mid bT,\ T \to bT \mid \lambda$$

is $\boldsymbol{a}^*\boldsymbol{b}^+$.

The language of

$$S \to aSb \mid \lambda$$

is $\{a^n b^n \mid n \geq 0\}$.

5 Derivation trees

Any derivation $S \overset{*}{\Rightarrow} w$ has a tree associated with it. The root of this tree is labelled S. At each individual production $A \to x$, make the symbols of x the children of the node corresponding to A in left to right order.

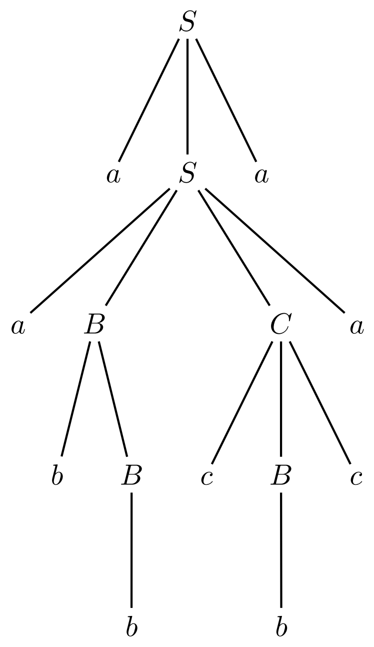
For example, consider the grammar:

$$S \to aSa \,|\, aBCa,\ B \to bB \,|\, b,\ C \to cCc \,|\, cBc$$

and the derivation:

$$S \Rightarrow aSa \Rightarrow aaBCaa \Rightarrow aabBCaa \Rightarrow aabBcBcaa \Rightarrow aabbcBcaa \Rightarrow aabbcbcaa$$

then the corresponding derivation tree is:



6 Regular grammars

A context free grammar is called a *regular grammar* if ever rule has one of the three forms:

$$A \to a,\ A \to aB,\ A \to \lambda$$

where in the second form $a \in \Sigma$, $B \in V$ (and $B = A$ is allowed).

Clearly in any derivation from S in a regular grammar there will only ever be at most one variable symbol produced, and it will always be rightmost in the current word.

From the name it's clear that the intention is that regular languages should be produced from regular grammars (and presumably vice versa) but we don't yet have the mechanics which will allow us to prove that conveniently (it *could* be done from the definitions, but we will shortly be introducing another way of generating regular languages and it will be most convenient to show that all three are equivalent to one another).

7 Regular grammars and NFAs

Recall that the rules of a regular grammar are of the form:

$$A \to cB,\ A \to c,\ A \to \lambda.$$

We can actually eliminate rules of the second type by introducing a special state Z, replacing them by $A \to cZ$, and having the only rule for Z be $Z \to \lambda$. Having done that, we can immediately construct an NFA that accepts the same words that a regular grammar generates. Namely, we take its states to be the non-terminals, its initial state to be S, its final states to be all those non-terminals for which there is a rule $X \to \lambda$ and its transitions to be $A \xrightarrow{c} B$ whenever there is a rule $A \to cB$ of the grammar.

We can equally well produce a regular grammar from an NFA and so we obtain:

Theorem 7.1. *The collection of languages generated by regular grammars, and the collection of languages accepted by NFAs (or DFAs) are the same.*

8 Regular languages and NFAs

We know that every regular language is accepted by some NFA. We would like to show that the language accepted by an NFA is always regular. This is accomplished by a reduction technique which generalizes the transitions allowed in an NFA – instead of having transitions labelled by single letters, we'll allow them to be labelled by regular languages.

So, begin with an NFA, M, and as usual assume that it has a unique start state and a unique distinct accepting state. For convenience we'll also add λ-transitions from each state, other than the initial and accepting states, of M to itself. We will construct a sequence of smaller and smaller NFAs with "extended transitions" that accept the same language as M does. Suppose that we've done this up to some point and there are still states other than the start and finish states remaining. Choose any such state q. Now consider every other pair of states r and s. If there are transitions:

$$r \overset{\boldsymbol{u}}{\to} q \quad \text{and} \quad q \overset{\boldsymbol{v}}{\to} s$$

then we will add a new transition $r \to s$ as follows, let $\boldsymbol{w}$ be the language on the loop on q and add:

$$r \xrightarrow{\boldsymbol{u}\boldsymbol{w}^*\boldsymbol{v}} s.$$

Having done this (for every pair), delete q. If we wind up with multiple transitions between any two states replace them by a single transition whose label is the union of their labels.

If we carry on like this we will eventually wind up with a single transition from the initial state to the accepting state. Its label is the language accepted by M. Since the steps we used in building labels are steps that are allowed in building regular languages, and since the initial labels are all basic regular languages, $L(M)$ must be regular. Thus we have proved:

Theorem 8.1. *The collections:*

- *regular languages,*
- *languages accepted by NFAs*

- *languages accepted by DFAs*
- *languages generated by a regular grammar*

are all the same.

9 Closure properties of regular languages

From the definition of regular language we know that regular languages are closed under union, concatenation, and Kleene star. However, we can use the preceding theorem to give us a much richer class of closure operations (i.e. many more ways of generating regular languages from simpler ones).

- The complement of a regular language is regular. Because, we can take a DFA accepting the language and then interchange all accepting and non-accepting states. The new automaton accepts the complement.
- The intersection of two regular languages is regular. Because:

  $$L_1 \cap L_2 = (L_1^c \cup L_2^c)^c$$

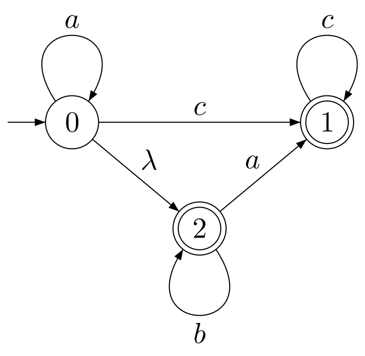
  and we already have closure under complement and union.
- If L is a regular language, then the collection of all suffixes (or prefixes) of words in L is also a regular language. Because, we can take a DFA for L which we can assume has the property that every state can be reached somehow from the initial state (if there were inaccessible states we could just throw them all away without changing the language). Now add a λ-transition from the initial state to every other state. The resulting automaton accepts suffixes of words in L. A similar construction works for prefixes.
- If L is a regular language, then so is L^r the language of all reversals of words in L. This is probably most easily proven inductively from the recursive definition. It's certainly true for the basic languages and thereafter we have $(L_1 L_2)^r = L_2^r L_1^r$, $(L_1 \cup L_2)^r = L_1^r \cup L_2^r$ and $(L^*)^r = (L^r)^*$.

We could extend this list even further, but the properties above are among the most useful. This shows that the regular languages form a very robust set – we can build lots and lots of them. And yet we believe that some simple languages like $\{a^n b^n \mid n \geq 0\}$ are not regular. It's time to introduce some techniques for proving such results.

10 Tutorial problems

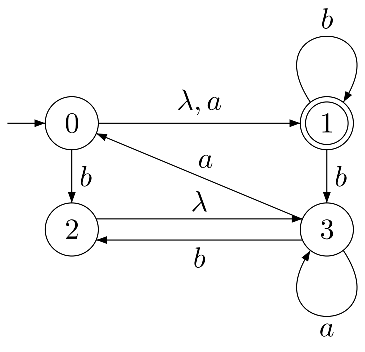
“The faster you derive, the bigger the mess”

1. Let M be the NFA



   - Compute the λ-closure of each state.
   - Construct a DFA that is equivalent to M
   - Give a regular expression for $L(M)$.

2. Repeat the preceding question with the following automaton:



3. Let G be the grammar:

$$\begin{aligned} S &\rightarrow abSc \,|\, A \\ A &\rightarrow cAd \,|\, cd \end{aligned}$$

   (a) Give a derivation of $ababccddcc$ and build its derivation tree.

   (b) Use set notation to describe $L(G)$.

4. Design context free grammars for the following languages (the alphabet is $\{a, b\}$ throughout). Where possible, try to design regular grammars for the same languages.

   (a) $L_1 = \{a^{4n} \,|\, n > 0\} \cup \{a^{3n+2} \,|\, n \geqslant 0\}$.

   (b) The language, PALINDROME, consisting of all strings that read the same forwards as backwards.

   (c) The language of strings that contain at least one occurrence of aa as a substring.

   (d) The language, EQUAL, consisting of all strings that contain the same number of a's as b's.

   (e) The language EVEN-EVEN consisting of all strings that contain both an even number of a's and an even number of b's