Poetry

The Pumping Lemma

Your questions?

Any regular language L has a magic number p
And any long-enough word in L has the following property:
Amongst its first p symbols is a segment you can find
Whose repetition or omission leaves x amongst its kind.

So if you find a language L which fails this acid test,
And some long word you pump becomes distinct from all the rest,
By contradiction you have shown that language L is not
A regular guy, resiliant to the damage you have wrought.

But if, upon the other hand, x stays within its L,
Then either L is regular, or else you chose not well.
For w is xyz, and y cannot be null,
And y must come before p symbols have been read in full.

As mathematical postscript, an addendum to the wise:
The basic proof we outlined here does certainly generalize.
So there is a pumping lemma for all languages context-free,
Although we do not have the same for those that are r.e.

-- Martin Cohn

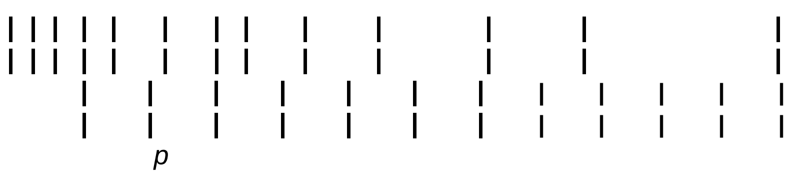
$L = \{a^n: n \text{ is prime}\}$

$L = \{w = a^n: n \text{ is prime}\}$

Let $w = a^j$, where j = the next prime number greater than k:

$$\underbrace{a\ a\ a\ a\ a\ a\ a}_{x}\ \underbrace{a\ a\ a}_{y}\ \underbrace{a\ a\ a}_{z}$$

- $|x| + |z|$ may be prime.
- $|x| + |y| + |z|$ is prime.
- $|x| + 2|y| + |z|$ may be prime.
- $|x| + 3|y| + |z|$ may be prime, and so forth.



But the Prime Number Theorem tells us that the primes "spread out", i.e., that the number of primes not exceeding x is asymptotic to $x/\ln x$.

$L = \{a^n : n \text{ is prime}\}$

Let $w = a^j$, where j is the smallest prime number $> k+1$.

$y = a^p$ for some p.

$\forall q \geq 0$ ($a^{|x| + |z| + q|y|}$ must be in L). So $|x| + |z| + q \cdot |y|$ must be prime.

But suppose that $q = |x| + |z|$. Then:

$$\begin{aligned} |x| + |z| + q \cdot |y| &= |x| + |z| + (|x| + |z|) \cdot y \\ &= (|x| + |z|) \cdot (1 + |y|), \end{aligned}$$

which is non-prime if both factors are greater than 1:

$L = \{a^n : n \text{ is prime}\}$

Let $w = a^j$, where j is the smallest prime number $> k+1$.

$y = a^p$ for some p.

$\forall q \geq 0$ ($a^{|x| + |z| + q|y|}$ must be in L). So $|x| + |z| + q \cdot |y|$ must be prime.

But suppose that $q = |x| + |z|$. Then:

$$\begin{aligned} |x| + |z| + q \cdot |y| &= |x| + |z| + (|x| + |z|) \cdot y \\ &= (|x| + |z|) \cdot (1 + |y|), \end{aligned}$$

which is non-prime if both factors are greater than 1:

$(|x| + |z|) > 1$ because $|w| > k+1$ and $|y| \leq k$.
$(1 + |y|) > 1$ because $|y| > 0$.

$L = \{a^i b^j : i, j \geq 0 \text{ and } i \neq j\}$

Try to use the Pumping Theorem by letting $w = a^{k+1}b^k$:

$L = \{a^i b^j : i, j \geq 0 \text{ and } i \neq j\}$

Try to use the Pumping Theorem by letting $w = a^k b^{k+k!}$.

Then $y = a^p$ for some nonzero p.

Let $q = (k!/p) + 1$ (i.e., pump in $(k!/p)$ times).

Note that $(k!/p)$ must be an integer because $p \leq k$.

The number of a's in the new string is $k + (k!/p)p = k + k!$.

So the new string is $a^{k+k!}b^{k+k!}$, which has equal numbers of a's and b's and so is not in L.

$L = \{a^i b^j : i, j \geq 0 \text{ and } i \neq j\}$

An easier way:

If L is regular then so is $\neg L$. Is it?

$L = \{a^i b^j : i, j \geq 0 \text{ and } i \neq j\}$

An easier way:

If L is regular then so is $\neg L$. Is it?

$\neg L = A^n B^n \cup$ {out of order}

If $\neg L$ is regular, then so is $L' = \neg L \cap a^*b^*$

$= \_\_\_\_\_\_\_\_\_\_\_\_$

$L = \{ a^i b^j c^k : i, j, k \geq 0$ and $(\text{if } i=1 \text{ then } j=k) \}$

This is example 8.16 in the textbook. Be sure to read it.

Using the Pumping Theorem Effectively

- To choose *w*:
  - Choose a *w* that is in the part of *L* that makes it not regular.
  - Choose a *w* that is only barely in *L*.
  - Choose a *w* with as homogeneous as possible an initial region of length at least *k*.
- To choose *q*:
  - Try letting *q* be either 0 or 2.
  - If that doesn't work, analyze *L* to see if there is some other specific value that will work.

Regular Languages closed under chop?

Let $chop(L) =$
$\{w : \exists x \in L$
$(x = x_1 c x_2,$
$x_1 \in \Sigma_L^*,$
$x_2 \in \Sigma_L^*,$
$c \in \Sigma_L,$
$|x_1| = |x_2|$, and
$w = x_1 x_2)\}$

What do we need to do:

a. If the answer is yes?

b. If the answer is no?

Also see Examples 8.20(firstchars), 8.22(maxstring, 8.23(mix)

Is the set of regular languages closed under *chop*?

L	*chop*(*L*)
$\varnothing$	
a*b*	
a*db*	

Decision Procedures

A decision procedure is an algorithm whose result is a Boolean value. It must:

- Halt
- Be correct

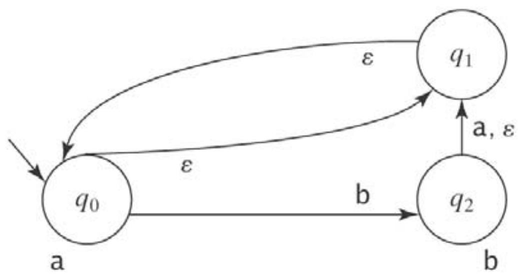
Important decision procedures exist for regular languages:

- Given an FSM *M* and a string *s*, does *M* accept *s*?
- Given a regular expression α and a string *w*, does α generate *w*?

Membership

We can answer the membership question by running an FSM.

But we must be careful if it's an NDFSM:



Membership

decideFSM(FSMdescription <M>, string w)
If *ndfsmsimulate*(*M*, *w*) accepts then return *True*
else return *False*.

Recall that ndfsmsimulate takes epsilon-closure at every stage, so there is no danger of getting into an infinite loop.

decideregex(regex α, string w)
From α, use *regextofsm* to construct an FSM *M* such that $L(\alpha) = L(M)$.
Return *decideFSM*(*M*, *w*).

Emptiness and Finiteness

- Given an FSM M, is $L(M)$ empty?
- Given an FSM M, is $L(M) = \Sigma_M*$?
- Given an FSM M, is $L(M)$ finite?
- Given an FSM M, is $L(M)$ infinite?
- Given two FSMs M_1 and M_2, are they equivalent?

Emptiness

- Given an FSM M, is $L(M)$ empty?
  - The graph analysis approach:
    1. Mark all states that are reachable via some path from the start state of M.
    2. If at least one marked state is an accepting state, return *False*. Else return *True*.
  - The simulation approach:
    1. Let M' = *ndfsmtodfsm*(M).
    2. For each string w in Σ^* such that $|w| < |K_{M'}|$ do:
       Run *decideFSM*(M', w).
    3. If M' accepts at least one such string, return *False.* Else return *True*.
  - The minimal DFSM approach:
    1. Create a minimal DFSM M′ that is equivalent to M.
    2. If M′ has exactly one state that is not Accepting , return *True*.
       Else return *False*.

Totality

- Given an FSM M, is $L(M) = \Sigma_M^*$?

Finiteness

- Given an FSM M, is $L(M)$ finite?
- The graph analysis approach:
- The simulation approach

Equivalence

- Given two FSMs M_1 and M_2, are they equivalent? In other words, is $L(M_1) = L(M_2)$? We can describe two different algorithms for answering this question.

Equivalence

- Given two FSMs M_1 and M_2, are they equivalent? In other words, is $L(M_1) = L(M_2)$?

equalFSMs$_1$(M_1: FSM, M_2: FSM) =

1. M_1' = *buildFSMcanonicalform*(M_1).
2. M_2' = *buildFSMcanonicalform*(M_2).
3. If M_1' and M_2' are equal, return *True*, else return *False*.

Equivalence

- Given two FSMs M_1 and M_2, are they equivalent? In other words, is $L(M_1) = L(M_2)$?

Observe that M_1 and M_2 are equivalent iff:

$$(L(M_1) - L(M_2)) \cup (L(M_2) - L(M_1)) = \varnothing.$$

equalFSMs$_2$(M_1: FSM, M_2: FSM) =

1. Construct M_A to accept $L(M_1) - L(M_2)$.
2. Construct M_B to accept $L(M_2) - L(M_1)$.
3. Construct M_C to accept $L(M_A) \cup L(M_B)$.
4. Return *emptyFSM*(M_C).

Minimality

- Given DFSM M, is M minimal?

Answering Specific Questions

Given two regular expressions α_1 and α_2, is:

$$(L(\alpha_1) \cap L(\alpha_2)) - \{\varepsilon\} \neq \varnothing?$$

1. From α_1, construct an FSM M_1 such that $L(\alpha_1) = L(M_1)$.
2. From α_2, construct an FSM M_2 such that $L(\alpha_2) = L(M_2)$.
3. Construct M' such that $L(M') = L(M_1) \cap L(M_2)$.
4. Construct M_ε such that $L(M_\varepsilon) = \{\varepsilon\}$.
5. Construct M'' such that $L(M'') = L(M') - L(M_\varepsilon)$.
6. If $L(M'')$ is empty return *False*; else return *True*.

(in the exercises, you'll write an algorithm for step 6)

Summary of Algorithms

- Operate on FSMs without altering the language that is accepted:
  - *Ndfsmtodfsm*
  - *MinDFSM*

Summary of Algorithms

- Compute functions of languages defined as FSMs:
  - Given FSMs M_1 and M_2, construct a FSM M_3 such that
    $L(M_3) = L(M_2) \cup L(M_1)$.
  - Given FSMs M_1 and M_2, construct a new FSM M_3 such that
    $L(M_3) = L(M_2)\, L(M_1)$.
  - Given FSM M, construct an FSM M^* such that
    $L(M^*) = (L(M))^*$.
  - Given a DFSM M, construct an FSM M^* such that
    $L(M^*) = \neg L(M)$.
  - Given two FSMs M_1 and M_2, construct an FSM M_3 such that
    $L(M_3) = L(M_2) \cap L(M_1)$.
  - Given two FSMs M_1 and M_2, construct an FSM M_3 such that
    $L(M_3) = L(M_2) - L(M_1)$.
  - Given an FSM M, construct an FSM M^* such that
    $L(M^*) = (L(M))^R$.
  - Given an FSM M, construct an FSM M^* that accepts
    letsub($L(M)$).

Algorithms, Continued

- Converting between FSMs and regular expressions:
  - Given a regular expression α, construct an FSM M such that:
    $$L(\alpha) = L(M)$$
  - Given an FSM M, construct a regular expression α such that:
    $$L(\alpha) = L(M)$$
- Algorithms that implement operations on languages defined by regular expressions: any operation that can be performed on languages defined by FSMs can be implemented by converting all regular expressions to equivalent FSMs and then executing the appropriate FSM algorithm.

Algorithms, Continued

- Converting between FSMs and regular grammars:
  - Given a regular grammar G, construct an FSM M such that:
    $$L(G) = L(M)$$
  - Given an FSM M, construct a regular grammar G such that:
    $$L(G) = L(M).$$

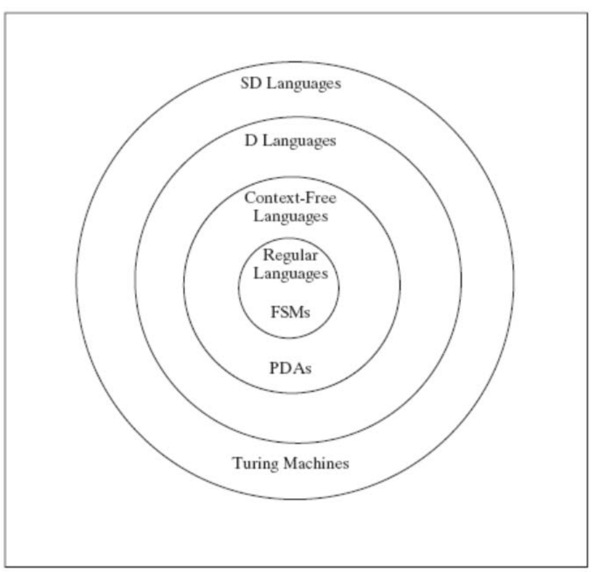
Algorithms: Decision Procedures

- Decision procedures that answer questions about languages defined by FSMs:
  - Given an FSM M and a string s, decide whether s is accepted by M.
  - Given an FSM M, decide whether $L(M)$ is empty.
  - Given an FSM M, decide whether $L(M)$ is finite.
  - Given two FSMs, M_1 and M_2, decide whether $L(M_1) = L(M_2)$.
  - Given an FSM M, is M minimal?
- Decision procedures that answer questions about languages defined by regular expressions: Again, convert the regular expressions to FSMs and apply the FSM algorithms.

Context-Free Grammars

Chapter 11

Languages and Machines



Rewrite Systems and Grammars

A ***rewrite system*** (or ***production system*** or ***rule-based system***) is:

- a list of rules, and
- an algorithm for applying them.

Each rule has a left-hand side and a right hand side.

Example rules:

$S \rightarrow aSb$
$aS \rightarrow \varepsilon$
$aSb \rightarrow bSabSa$

Simple-rewrite

simple-rewrite(R: rewrite system, w: initial string) =

1. Set *working-string* to w.

2. Until told by R to halt do:
   Match the lhs of some rule against some part of *working-string*.

   Replace the matched part of *working-string* with the rhs of the rule that was matched.

3. Return *working-string*.

A Rewrite System Formalism

A rewrite system formalism specifies:

- The form of the rules
- How *simple-rewrite* works:
  - How to choose rules?
  - When to quit?

An Example

$w = SaS$

Rules:

[1] $S \rightarrow aSb$
[2] $aS \rightarrow \varepsilon$

- What order to apply the rules?
- When to quit?

Rule Based Systems

- Expert systems
- Cognitive modeling
- Business practice modeling
- General models of computation
- **Grammars**

Grammars Define Languages

A grammar, G, is a set of rules that are stated in terms of two alphabets:

• a ***terminal alphabet***, Σ, that contains the symbols that make up the strings in $L(G)$, and

•a ***nonterminal alphabet***, the elements of which will function as working symbols that will be used while the grammar is operating. These symbols will disappear by the time the grammar finishes its job and generates a string.

A grammar has a unique start symbol, often called S.

Using a Grammar to Derive a String

Simple-rewrite (G, S) will generate the strings in $L(G)$.

We will use the symbol $\Rightarrow$ to indicate steps in a derivation.

In our example:

[1] $S \rightarrow aSb$
[2] $aS \rightarrow \varepsilon$

A derivation could begin with:

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow \ldots$

Generating Many Strings

- Multiple rules may match.

Given: $S \rightarrow aSb$, $S \rightarrow bSa$, and $S \rightarrow \varepsilon$

Derivation so far: $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow$

Three choices at the next step:

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb$ (using rule 1),
$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabSabb$ (using rule 2),
$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$ (using rule 3).

Generating Many Strings

- One rule may match in more than one way.

Given: $S \rightarrow \mathrm{a}TT\mathrm{b}$, $T \rightarrow \mathrm{b}T\mathrm{a}$, and $T \rightarrow \varepsilon$

Derivation so far: $S \Rightarrow \mathrm{a}TT\mathrm{b} \Rightarrow$

Two choices at the next step:

$S \Rightarrow \mathrm{a}\underline{T}T\mathrm{b} \Rightarrow \mathrm{ab}T\mathrm{a}T\mathrm{b} \Rightarrow$
$S \Rightarrow \mathrm{a}T\underline{T}\mathrm{b} \Rightarrow \mathrm{a}T\mathrm{b}T\mathrm{ab} \Rightarrow$

When to Stop

May stop when:

1. The working string no longer contains any nonterminal symbols (including, when it is ε).

In this case, we say that the working string is ***generated*** by the grammar.

Example:

$S \Rightarrow \mathrm{a}S\mathrm{b} \Rightarrow \mathrm{aa}S\mathrm{bb} \Rightarrow \mathrm{aabb}$

When to Stop

May stop when:

2. There are nonterminal symbols in the working string but none of them is in a substring that is the left-hand side of any rule in the grammar.

In this case, we have a blocked or non-terminated derivation but no generated string.

Example:

Rules: $S \rightarrow aSb$, $S \rightarrow bTa$, and $S \rightarrow \varepsilon$

Derivations: $S \Rightarrow aSb \Rightarrow abTab \Rightarrow$ [blocked]

When to Stop

It is possible that neither (1) nor (2) is achieved.

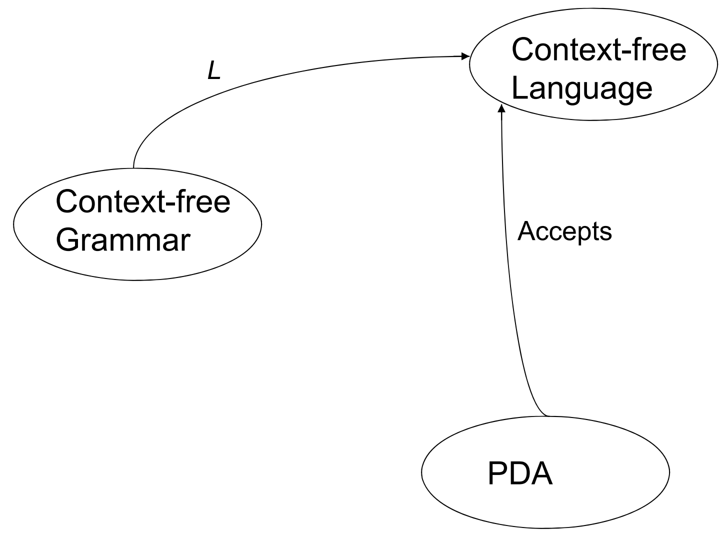
Example:

G contains only the rules $S \rightarrow Ba$ and $B \rightarrow bB$, with S the start symbol.

Then all derivations proceed as:

$$S \Rightarrow Ba \Rightarrow bBa \Rightarrow bbBa \Rightarrow bbbBa \Rightarrow bbbbBa \Rightarrow \ldots$$

Context-free Grammars, Languages, and PDAs



Context-Free Grammars

No restrictions on the form of the right hand sides.

$S \rightarrow \mathrm{ab}DeFG\mathrm{ab}$

But require single non-terminal on left hand side.

$S \rightarrow$

but not $ASB \rightarrow$

$a^n b^n$

Balanced Parentheses language

$a^m b^n : m >= n$

Context-Free Grammars

A context-free grammar G is a quadruple,
(V, Σ, R, S), where:

- V is the rule alphabet, which contains nonterminals and terminals.
- Σ (the set of terminals) is a subset of V,
- R (the set of rules) is a finite subset of $(V - \Sigma) \times V^*$,
- S (the start symbol) is an element of $V - \Sigma$.

Example:

$(\{S, a, b\}, \{a, b\}, \{S \rightarrow a\,S\,b, S \rightarrow \varepsilon\}, S)$

Rules are also known as **productions**.