The Pumping Lemma, Context Free Grammars

CS154

Chris Pollett

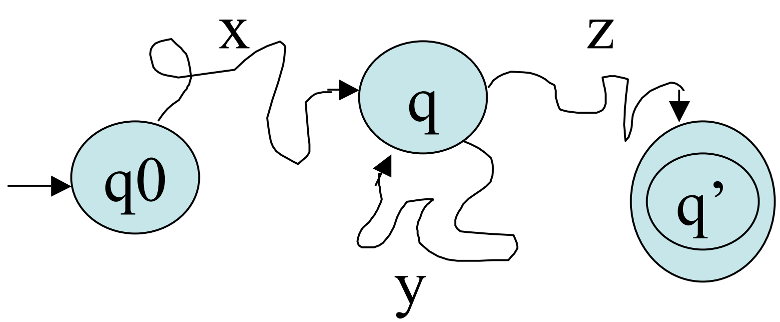
Feb 26, 2007.

Outline

- The Pumping Lemma
- Context free Grammars

The Pumping Lemma

- Suppose we have a machine M with k states. Feed in some input string w of length n>k. At some point in the computation, by the Pigeonhole principle, the machine must repeat a state.
- Suppose M accepts w. Then can imagine M's computation splitting w into 3 pieces, w=xyz, according to the diagram:



More on the Pumping Lemma

But this implies that M accepts the strings xz, xyyz, xyyyz, etc.

This is essentially what the Pumping Lemma says:

Lemma (Pumping Lemma).

If A is a regular language, then there is a number p (the pumping length) where, if s is any string in A of length at least p, then s may be divided into three pieces s=xyz, such that:

for each i>= 0, xy^iz is in A

$|y| > 0$, and

$|xy| <= p$

Using the Pumping Lemma

- We can use the pumping lemma to show language are not regular.
- For example, let C={ w| w has an equal number of 0's and 1's}. To prove C is not regular:
  - Suppose DFA M that recognizes C.
  - Let p be M's pumping length
  - Consider the string $w = 0^p1^p$. This string is in the language and has length > p.
  - So by the pumping lemma w = xyz, where $|xy| \leq p$, $|y| > 0$, and where xy^iz is in the language for all $i \geq 0$. That means $x = 0^k$ and $y=0^j$ where $k+j \leq p$ and $j>0$. But then taking i=0, $xz = 0^{p-j}1^p$ should be in C. As p-j is not equal to p this give a contradiction. So C is not regular.

More Examples

Show $L = \{ww^R \mid w \in \Sigma^*\}$ is not regular.

- Suppose M is a DFA that recognizes L.
- Let p be M's pumping length
- Consider the string $w = 0^p110^p$. This string is in the language and has length > p.
- So by the pumping lemma w = xyz, where $|xy| \leq p$, $|y| > 0$, and where xy^iz is in the language for all $i \geq 0$. That means $x = 0^k$ and $y=0^j$ where $k+j \leq p$ and $j>0$. But then taking i=0, $xz = 0^{p-j}110^p$ should be in L. The two 11's not occur on the left hand half of xz and there are no 1's on the right hand half. So xz is not of the form string followed by reverse of the same string so in not in L, contradicting the pumping lemma. So L is not regular.

Show that $L = \{w \in \Sigma^* \mid n_a(w) < n_b(w)\}$ is not regular.

- Suppose M is a DFA that recognizes L.
- Let p be M's pumping length
- Consider the string $w = a^pb^{p+1}$. This string is in the language and has length > p.
- So by the pumping lemma w = xyz, where $|xy| \leq p$, $|y| > 0$, and where xy^iz is in the language for all $i \geq 0$. That means $x = a^k$ and $y=a^j$ where $k+j \leq p$ and $j>0$. But then taking i=2, $xy^2z = a^{p+j}b^{p+1}$ should be in L. As $j>0$, $n_a(xy^2z) = p+j$ is not less than $n_b(xy^2z) = p+1$. So xy^2z is not in L, contradicting the pumping lemma. So L is not regular.

Context Free Languages

- We saw that regular languages were useful for doing things like string matching.
- This might occur in practice as the so-called lexical analysis phase of compiler. That is, the phase in which we recognize tokens like language reserved words, variable names, constants, etc.
- We now turn to ways of specify programming languages or even aspects of natural languages.
- The key to this is to have some way to recognize the underlying structures such as nouns and verbs, or control blocks, etc of the language.
- Context Free Grammars (CFGs), which are a less restricted form of grammar than a regular grammar, and their languages will provide us with the tools to do this.

Example CFG

- Recall a grammar consists of a collection of **substitution rules** (aka **productions**). For instance:

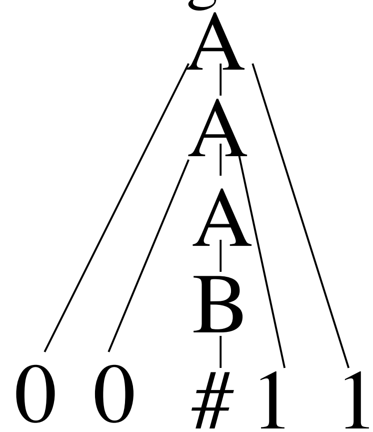
  A --> 0A1

  A --> B

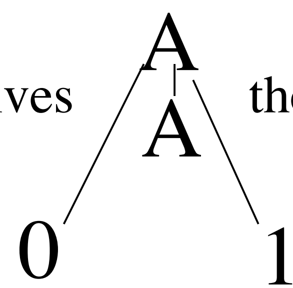
  B --> #
- A rule has a two types of symbols **variables** and **terminals**.
- Usually, we'll write variables using uppercase letters or in brackets like <variable>. Terminals are supposed to be strings over the alphabet of the language we are considering.
- In a CFG, the left hand side of each rule has one variable; the right hand side can be a string of variables and terminals.
- Variables can be substituted for; terminals cannot. One variable usually denoted by S is usually distinguishes as a **start variable**.
- An example sequence of substitutions (aka a **derivation**) in the above grammar might be: A => 0A1 => 00A11 => 00B11 => 00#11

More on CFGs

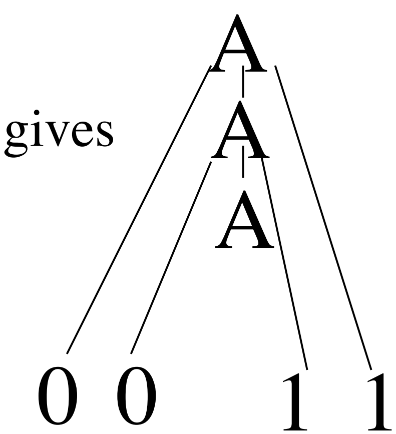
- Such a derivation might also be drawn as a **parse tree**:



- Here A=> 0A1 gives



then A==> 0A1 => 00A11 gives



etc.

Formal Definitions

- A **context free grammar** is a 4-tuple (V, Σ, R, S) where
  1. V is a finite set called the **variables**
  2. Σ is a finite set, disjoint from V called the **terminals**.
  3. R is a finite set of **rules**, with each rule being a pair consisting of a variable and a string of variables and terminals, and
  4. $S \in V$ is a start variable.
- For a rule A--> w where w is a string over $(V \cup \Sigma)$, and for other strings u and v, we say uAv **yields** uwv, written uAv => uwv. We say u derives v, written $u \Rightarrow^* v$, if there is a finite sequence:

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k \Rightarrow v.$$

- The language of a CFG is the set of of string over Σ^* derivable from its start symbol.
- A language given by a context free grammar is called a **context free language**.
- Sometimes we abbreviate multiple rules with same left hand side using a '|'. For example, A--> 0A1 | B .

Example

- Consider the grammar G= (V, Σ, R, <EXPR>) where V is
  {<EXPR>, <TERM>, <FACTOR>}
  and Σ is (a, +, x, (,))
  and the rules are:
  <EXPR> --> <EXPR> + <TERM> | <TERM>
  <TERM> --> <TERM> x <FACTOR> | <FACTOR>
  <FACTOR> --> (<EXPR>) | a
- One can verify that <EXPR> =>* (a+a) x a.
  - This is true since <EXPR> => <TERM> => <TERM> x <FACTOR> => <FACTOR> x <FACTOR> => (<EXPR>) x <FACTOR> => (<EXPR> + <TERM>) x <FACTOR> => (<TERM> + <TERM>) x <FACTOR> => (<FACTOR> + <TERM>) x <FACTOR> => (a+ <TERM>) x <FACTOR> => (a+ <FACTOR>) x <FACTOR> =>(a+a) x <FACTOR> => (a+a) x a

Techniques for Designing CFGs

- Many CFLs are the union of simpler CFLs. So one can design a CFG for each in turn with start states S_1, S_2,… S_n .Then take the union of the rules and add a new start variable with a rule $S \rightarrow S_1 | S_2 | \ldots | S_n$. For example, take the language $\{0^n 1^n \mid n >= 0\} \cup \{1^n 0^n \mid n >= 0\}$. First we could make CFGs for each language separately. Say, $S_1 \rightarrow 0\ S_1\ 1 | \varepsilon$ and $S_2 \rightarrow 1\ S_2\ 0 | \varepsilon$. Then add the rule $S \rightarrow S_1 | S_2$.
- Notice any regular grammar is already a CFG, so the regular languages are all CFGs.

More Techniques for Designing CFGs

- For CFL which contain two substrings which are linked in the sense that a machine for such a language would need to remember information about one on the strings to verify information about the other substring, you might want to consider rules of the form R --> u R v. Here u and v should satisfy the property you are trying to verify.