Designing Context-free Grammars

Designing CFGs

- As with design of FA, design of CFGs requires creativity
  - It is even trickier

- Useful techniques:

Technique 1: Break down to simpler pieces

(Many CFLs are union of simpler CFLs)

Individual grammars can be merged into a grammar for the original language by combining their rules and then adding the new rule

$$S \longrightarrow S_1 \mid S_2 \ldots \mid S_k$$

where the variables S_i are the start variables for the individual grammars.

Example

For example, to get a grammar for the language $\{0^n1^n | n \geq 0\} \cup \{1^n0^n | n \geq 0\}$, first construct the grammar

$$S_1 \rightarrow 0S_11 \mid \varepsilon$$

for the language $\{0^n1^n | \, n \geq 0\}$ and the grammar

$$S_2 \rightarrow 1S_20 \mid \varepsilon$$

for the language $\{1^n0^n | \, n \geq 0\}$ and then add the rule $S \rightarrow S_1 \mid S_2$ to give the grammar

$$\begin{aligned} S &\rightarrow S_1 \mid S_2 \\ S_1 &\rightarrow 0S_11 \mid \varepsilon \\ S_2 &\rightarrow 1S_20 \mid \varepsilon. \end{aligned}$$

Designing CFGs

Technique 2: constructing a CFG for a language that happens to be regular is easy if you can first construct a DFA for that language.

You can convert any DFA into an equivalent CFG as follows:

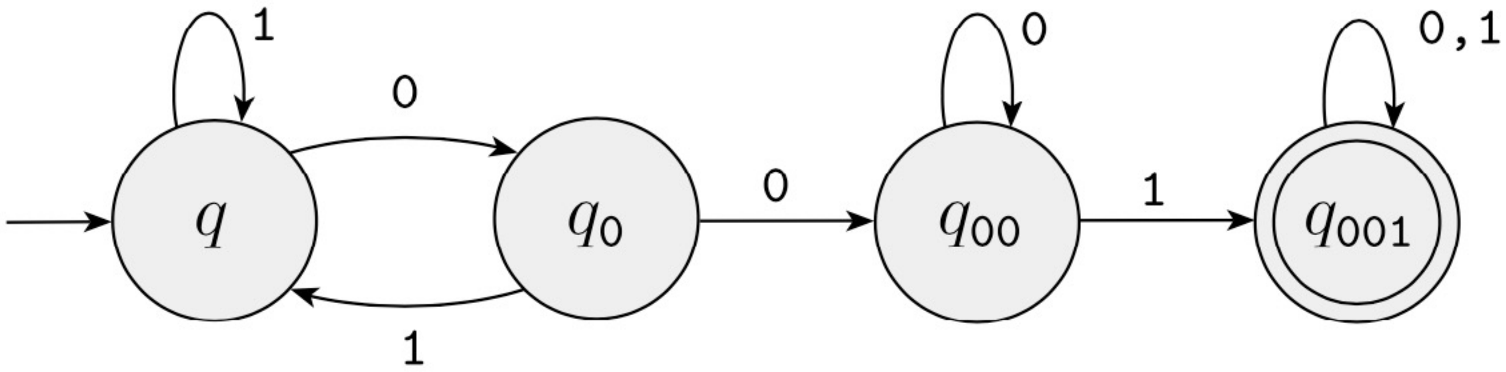
- *Mark a variable R_i for each state q_i of the DFA*
- *Add the rule $R_i \longrightarrow aR_j$ to the CFG if $\delta(q_i, a) = q_j$ is a transition in the DFA*
- *Add the rule $R_i \longrightarrow \varepsilon$ if q_i is an accept state of the DFA*
- *Make R_0 the start variable of the grammar, where q_0 is the start state of the machine*

Verify on your own that the resulting CFG generates the same language that the DFA recognizes

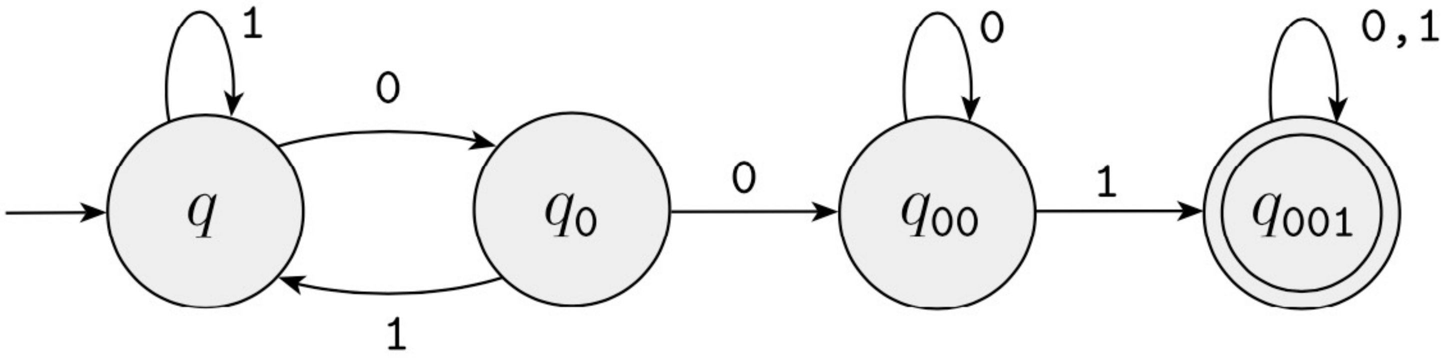
Example

Convert the regular language recognized by this DFA into a CFG.

First though, what is the language?



Example



$R \longrightarrow 0R_0 \mid 1R$

$R_0 \longrightarrow 0R_{00} \mid 1R$

$R_{00} \longrightarrow 0R_{00} \mid 1R_{001}$

$R_{001} \longrightarrow 0R_{001} \mid 1R_{001} \mid \varepsilon$

Designing CFGs

Technique 3: handle "links"

Certain CFLs contain strings with two substrings that are "linked" in the sense that a machine for such a language would need to remember an unbounded amount of information about one of the substrings to verify that it corresponds properly to the other substring.

<u>Example</u>: This situation occurs in the language $\{0^n1^n \mid n \geq 0\}$ because a machine would need to remember the number of 0s in order to verify that it equals the number of 1s

Designing CFGs

Technique 3: handle "links"

You can construct a CFG to handle this situation by using a rule of the form

$$R \longrightarrow uRv$$

which generates strings wherein the portion containing the u's corresponds to the portion containing the v's.

Designing CFGs

Technique 4: handle recursion

In more complex languages, the strings may contain certain structures that appear recursively as part of other (or the same) structures.

Example: this situation occurs in the grammar G_4 (arithmetic evaluations) we saw earlier.

Consider grammar $G_4 = (V, \Sigma, R, \langle\text{EXPR}\rangle)$.

V is $\{\langle\text{EXPR}\rangle, \langle\text{TERM}\rangle, \langle\text{FACTOR}\rangle\}$ and Σ is $\{\mathtt{a}, \mathtt{+}, \mathtt{x}, \mathtt{(}, \mathtt{)}\}$. The rules are

$$\begin{aligned}
\langle\text{EXPR}\rangle &\to \langle\text{EXPR}\rangle\mathtt{+}\langle\text{TERM}\rangle \mid \langle\text{TERM}\rangle \\
\langle\text{TERM}\rangle &\to \langle\text{TERM}\rangle\mathtt{x}\langle\text{FACTOR}\rangle \mid \langle\text{FACTOR}\rangle \\
\langle\text{FACTOR}\rangle &\to \mathtt{(}\,\langle\text{EXPR}\rangle\,\mathtt{)} \mid \mathtt{a}
\end{aligned}$$

Designing CFGs

Technique 4: handle recursion

Any time the symbol a appears, an entire parenthesized expression might appear recursively instead.

To achieve this effect, place the variable symbol generating the structure in the location of the rules corresponding to where that structure may recursively appear.

Ambiguity

- Sometimes a grammar can generate the same string in several different ways
- Such a string will have several different parse trees and thus several different meanings
- This is undesirable for certain applications, such as programming languages, where a program should have a unique interpretation.

- If a grammar generates the same string in several different ways, we say the string is derived ambiguously in that grammar.
- If a grammar generates some string ambiguously, we say that the grammar is ambiguous.

Ambiguity

- For example, consider the grammar G_5

$$\langle \text{EXPR} \rangle \rightarrow \langle \text{EXPR} \rangle + \langle \text{EXPR} \rangle \mid \langle \text{EXPR} \rangle \times \langle \text{EXPR} \rangle \mid (\langle \text{EXPR} \rangle) \mid \mathtt{a}$$

- G_5 generates the string a+a×a ambiguously, as shown below

Another example of an ambiguous grammar: the grammar G_2 we saw earlier

$$
\begin{aligned}
\langle \text{SENTENCE} \rangle &\to \langle \text{NOUN-PHRASE} \rangle \langle \text{VERB-PHRASE} \rangle \\
\langle \text{NOUN-PHRASE} \rangle &\to \langle \text{CMPLX-NOUN} \rangle \mid \langle \text{CMPLX-NOUN} \rangle \langle \text{PREP-PHRASE} \rangle \\
\langle \text{VERB-PHRASE} \rangle &\to \langle \text{CMPLX-VERB} \rangle \mid \langle \text{CMPLX-VERB} \rangle \langle \text{PREP-PHRASE} \rangle \\
\langle \text{PREP-PHRASE} \rangle &\to \langle \text{PREP} \rangle \langle \text{CMPLX-NOUN} \rangle \\
\langle \text{CMPLX-NOUN} \rangle &\to \langle \text{ARTICLE} \rangle \langle \text{NOUN} \rangle \\
\langle \text{CMPLX-VERB} \rangle &\to \langle \text{VERB} \rangle \mid \langle \text{VERB} \rangle \langle \text{NOUN-PHRASE} \rangle \\
\langle \text{ARTICLE} \rangle &\to \texttt{a} \mid \texttt{the} \\
\langle \text{NOUN} \rangle &\to \texttt{boy} \mid \texttt{girl} \mid \texttt{flower} \\
\langle \text{VERB} \rangle &\to \texttt{touches} \mid \texttt{likes} \mid \texttt{sees} \\
\langle \text{PREP} \rangle &\to \texttt{with}
\end{aligned}
$$

The sentence the girl touches the boy with the flower has two different derivations.

Formalizing the notion of ambiguity

- When we say that a grammar generates a string ambiguously, we mean that the string has two different *parse trees*, not two different *derivations*.
- Two derivations may differ merely in the order in which they replace variables, yet not in their overall structure.
- To concentrate on structure, we define a type of derivation that replaces variables in a **fixed** order.
- A derivation of a string w in a grammar G is a leftmost derivation if at every step the leftmost remaining variable is the one replaced.

Formalizing the notion of ambiguity

DEFINITION 2.7

A string w is derived ***ambiguously*** in context-free grammar G if it has two or more different leftmost derivations. Grammar G is ***ambiguous*** if it generates some string ambiguously.

Note: sometimes when we have an ambiguous grammar, we can find an unambiguous grammar that generates the same language (e.g G_5 vs G_4)

However, some CFLs can be generated only by ambiguous grammars.
Such languages are called ***inherently ambiguous***.
Example: $\{a^i\, b^j\, c^k \mid i = j \text{ or } j = k\}$