Formal Grammars

1. Last time, we looked informally at how context-free grammars could be used to describe languages and began to formalize this notation. Today, I want to review and complete the precise definitions we will use to interpret the meaning of a context-free grammar.

2. Formally:

**Context-free Grammar** A context free grammar is composed of:

(a) A finite alphabet \( V \) of variables (or non-terminals).
(b) A distinct finite alphabet \( \Sigma \) called the terminals or terminal symbols.
(c) \( S \in V \) referred to as the start symbol.
(d) A finite set \( R \) of pairs composed of one element from \( (V \cup \Sigma)^* \) called the rules. Rules are usually written in the form:

\[ A \rightarrow X_1X_2...X_m \]

rather than \( (A, X_1X_2...X_m) \).

3. The association between a context free grammar and the language it describes is formalized through the notion of a derivation:

**Yields** Given a grammar, \( G = (V, \Sigma, S, R) \), and two strings \( x \) and \( y \) in \( (V \cup \Sigma)^* \) we say that \( x \) derives \( y \) if there exists a sequence of string \( \alpha_0, \alpha_1, \alpha_2, ..., \alpha_m \) all in \( (V \cup \Sigma)^* \) such that

(a) for all \( i < m \), \( \alpha_i \Rightarrow \alpha_{i+1} \),
(b) \( x = \alpha_0 \), and
(c) \( y = \alpha_m \).

In this case we write

\[ x \Rightarrow y \]

The sequence \( \alpha_0, \alpha_1, \alpha_2, ..., \alpha_m \) is called a derivation of length \( m \) of \( y \) from \( x \).

4. Examples of direct derivations.

Consider \( G = (\{B,G\}, \{a,x,z\}, B, \{(B, xGBy), (B, z), (G, aG), (G, \epsilon)\}) \) or less formally

\[
\begin{align*}
B & \rightarrow x \ G \ B \ y \\
B & \rightarrow z \\
G & \rightarrow a \ G \\
G & \rightarrow \epsilon
\end{align*}
\]

The following are examples of the “yields” relation for this grammar:

- \( B \Rightarrow x \ G \ B \ y \)
- \( x \ G \ a \Rightarrow x \ a \ G \ a \)

5. More on the notion of a derivation:

**Derivation** Given a grammar, \( G = (V, \Sigma, S, R) \), and two strings \( x \) and \( y \) in \( (V \cup \Sigma)^* \) we say that \( x \) derives \( y \) if there exists a sequence of string \( \alpha_0, \alpha_1, \alpha_2, ..., \alpha_m \) all in \( (V \cup \Sigma)^* \) such that

(a) for all \( i < m \), \( \alpha_i \Rightarrow \alpha_{i+1} \),
(b) \( x = \alpha_0 \), and
(c) \( y = \alpha_m \).

In this case we write

\[ x \Rightarrow^* y \]

The sequence \( \alpha_0, \alpha_1, \alpha_2, ..., \alpha_m \) is called a derivation of length \( m \) of \( y \) from \( x \).

6. Using the grammar \( G \) shown above we can say that that \( B \Rightarrow^* xaxzyy \) since:

- \( B \Rightarrow x \ G \ B \ y \)
- \( x \ G \ B \ y \Rightarrow x \ a \ G \ B \ y \)
- \( x \ a \ G \ B \ y \Rightarrow x \ a \ B \ y \)
- \( x \ a \ B \ y \Rightarrow x \ a \ x \ G \ B \ y \ y \)
- \( x \ a \ x \ G \ B \ y \ y \Rightarrow x \ a \ x \ B \ y \ y \)
- \( x \ a \ x \ B \ y \ y \Rightarrow x \ a \ x \ z \ y \ y \)
7. Time for more definitions:

**Sentential form** Given a grammar, \( G = (V, \Sigma, S, R) \), a string \( w \) is called a sentential form of \( G \) if \( S \xrightarrow{*} w \).

**Sentence** A sentential form containing only symbols from the terminal vocabulary of a language is called a sentence.

**\( L(G) \)** The language defined by a grammar \( G \) is the set of all sentences.
\[
L(G) = \{ w \mid S \xrightarrow{*} w \text{ and } w \in \Sigma^* \}
\]

8. To make the advantages of this formalism a bit more apparent, consider how it can be used to clarify the arguments one might use to show that all regular languages are context-free.

9. For the DFA version, given \( D = (Q, \Sigma, \delta, s, F) \) we would construct a grammar \( G = (Q, \Sigma, s, R) \) where \( R = \{ (q, xq') \mid q' = \delta(q, x) \} \cup \{ (q, \epsilon) \mid q \in F \} \).

Then, we can show that \( \hat{\delta}(s, w) = q \) in \( D \) if and only if \( s \xrightarrow{*} wq \) in \( G \). The proof is by induction on the length of \( w \). For the basis, it is clear that for \( w = \epsilon \), \( \hat{\delta}(s, \epsilon) = s \) and \( s \xrightarrow{*} s \) as required.

Now, we assume the result holds for \( |w| < n \) and consider any string \( wx \) with \( x \in \Sigma \) and \( |wx| = n \). If \( \hat{\delta}(s, wx) = q \) then by the definition of \( \hat{\delta} \), \( \hat{\delta}(\hat{\delta}(s, w), x) = q \). Let \( q' = \hat{\delta}(s, w) \). Then, we know that \( \delta(q', x) = q \) so by our definition of \( G \), \( (q', xq) \in R \). By our inductive assumption, \( s \xrightarrow{*} wq' \). Therefore, by adding one step using the rule \( (q', xq) \) we can conclude that \( s \xrightarrow{*} wxq \). On the other hand, suppose we know that \( s \xrightarrow{*} wxq \). Given the way we have defined \( G \), we know that this derivation must end with the application of a rule of the form \( (q', xq) \in R \). That is, we can write \( s \xrightarrow{*} wq' \xrightarrow{*} wxq \). Such a rule, however, would only be included if \( \delta(q', x) = q \). Moreover, given that \( s \xrightarrow{*} wq' \) our inductive assumption implies that \( \hat{\delta}(s, w) = q' \). From this, we can conclude that \( \hat{\delta}(s, wx) = \hat{\delta}(\hat{\delta}(s, w), x) = \hat{\delta}(q', x) = q \) as desired.

Finally, we note that we can extend a derivation of the form \( S \xrightarrow{*} wq \xrightarrow{*} w \) to generate a sentence in \( G \) if and only if \( G \) contains a rule of the form \( (q, \epsilon) \in R \). \( G \) contains such a rule for \( q \) if and only if \( q \in F \) in \( D \). Therefore, \( S \xrightarrow{*} wq \xrightarrow{*} w \) if and only if \( w \in L(D) \).

---

### Pushdown Automata

1. We first encountered regular languages as those languages recognized by a class of automata (DFAs) and then encountered a generative notation (regular expressions) that was associated with exactly the same class of languages.

2. With context-free languages, we are taking the opposite approach. First, I presented a generative notation (context-free grammars) that describes these languages. Now, we will examine a type of automaton that recognizes the same set of languages.

3. The new type of automata is not finite! Instead, we will allow for a potentially infinite memory.

   - We will still limit our attention to machines that accept or reject finite input strings.
   - We will restrict access to the machine’s memory in a way that makes them more powerful than finite automata, but still limited in computational power.

4. The model we will consider next is called a pushdown automata.

   - The control of a pushdown automata will be a lot like a finite automata. The machine will have a finite set of states and a transition function that specifies the possible moves.
   - It is nondeterministic.
   - The control has final and non-final states and accepts if there is a way that it can be in a final state when the input is exhausted.
   - In addition to its input, the machine will have the ability to read and write symbols on an unbounded stack.
   - The symbol on the top of the stack can help determine the possible transitions from the current state and can be removed from the stack as part of the transition.
   - In addition to a new state, a transition can specify a new symbol that should be pushed onto the stack.
The alphabet used on the stack can be distinct from the alphabet of the input language.

5. An example should make all of this clear. Recall the language \( \{1^n = 1^n | n \geq 0 \} \).

- This was one of the early examples of a language we showed was not regular.
- Consider how this language can be accepted by a pushdown automata.
  - The diagram below provides an informal description of a pushdown automaton that recognizes this language.
  - When we draw a state diagram for a PDA, each edge is labeled by a triple “i, p/s” where i is the current input symbol, p is the symbol to be popped from the stack while taking the transition (or \( \epsilon \) if nothing needs to be popped while taking the transition), and s is either a symbol to be pushed on the stack or \( \epsilon \).
  - Basically, a pushdown automata can count up by sticking symbols on the stack as it reads one part of the input and count down by popping those symbols later.
  - Note that this machine has one feature that is somewhat an artifact of the way Sipser chooses to describe PDAs — namely, his formalism provides no way for the machine to sense if its stack is empty. This will lead most of our machines to include a start state with just one transition that puts a recognizable symbol at the bottom of the stack before processing any input.
  - Also note that this trick depends on epsilon-transitions. In particular, for now, all PDAs are non-deterministic.

6. Better yet, working some examples should make this even clearer, so...

- Please construct a PDA that recognizes

\[ L_{add} = \{1^i + 1^j = 1^{i+j} | i, j \geq 1 \} \]

Here is one solution:

- It is generally easier to construct a PDA for a language than to understand one that someone else constructed (particularly if they don’t help you out by providing a little prose explaining what the machine is supposed to do — Hint: Please pity the graders).

So, consider these two PDAs. Can you describe the languages they accept?
Actually, this is a trick question. Both PDAs accept the same language:

\[ L_{eq occur} = \{ w \mid w \in \{a,b\}^* \text{ and } w \text{ contains as many a's as b's} \} \]

The machine with the states named A and B was my first attempt and may be a bit easier to understand. The idea is that as one scanned the symbols of some input \( w \), one could plot the difference between the number of as and bs (i.e., \( \#(a) - \#(b) \)).

This value must start as 0 and end as 0 if \( w \) is in the language. It may be positive or negative in between. Between any point where it is positive and one where it is negative (or vice versa) it must equal 0 at some symbol. The idea behind this machine is that it will spend its time in state A while \( \#(a) - \#(b) \) is positive, spend its time in state B while \( \#(a) - \#(b) \) is negative, and have the option to non-deterministically switch between A and B when the number of as and bs seen so far is equal.

To ensure that the machine functions in this way, when in state A, it pushes as on the stack when it sees as and pops as when it sees b. That is, the number of as on the stack should equal \( \#(a) - \#(b) \) while in state A. When the machine is in state B, on the other hand, the goal is to have the number of bs on the stack equal \( \#(b) - \#(a) \).

The second machine simplifies (complicates?) the first by observing that if we are willing to assume that the non-determinism is smart enough to let the first machine know when to move from state A to state B, we can instead just assume the non-determinism can be smart enough to know when to push as or use them to pops bs and vice versa.

**PDAs Formally**

1. A *pushdown automaton* is a 6-tuple \( (Q, \Sigma, \Gamma, \delta, q_0, F) \) where:
   - \( Q \) is a finite set of states,
   - \( \Sigma \) is a finite input alphabet,
   - \( \Gamma \) is a finite stack alphabet,
   - \( \delta : Q \times \Sigma \times \Gamma \rightarrow P(Q \times \Gamma) \) is the transition function, and
   - \( F \subset Q \) is the set of final or accepting states.

2. We say that a PDA \( M = (Q, \Sigma, \Gamma, \delta, q_0, F) \) accepts a string \( w = w_1w_2\ldots w_n \), \( w_i \in \Sigma \) if \( \exists q_1, \ldots q_n \in Q \) and \( s_0, s_1, \ldots s_n \in \Gamma^* \) such that:
   - \( s_0 = \epsilon \)
   - \( \forall i, 1 \leq i \leq n, \exists h_i, p_i \in \Gamma \) and \( t_i \in \Gamma^* \) such that \( s_{i-1} = h_it_i, s_i = p_it_i \) and \( (q_i, p_i) \in \delta(q_{i-1}, w_i, h_i) \)
   - \( s_n \in F \)