Definition of GNFA

A generalized non-deterministic finite automaton is a 5-tuple

\((Q, \Sigma, \delta, q_0, F)\)

where

1. \(Q\) is a finite set of states
2. \(\Sigma\) is a finite alphabet
3. \(\delta: (Q - \{q_{\text{accept}}\}) \times (Q - \{q_{\text{start}}\}) \rightarrow \mathbb{R}\) is the transition function
4. \(q_{\text{start}} \in Q\) is the start state
5. \(q_{\text{accept}} \in Q\) is the accept state and \(q_{\text{start}} \neq q_{\text{accept}}\)

A GNFA accepts a string \(w \in \Sigma^*\) if \(w = w_1w_2...w_k\) where each \(w_i \in \Sigma^*\) and a sequence of states \(q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow ... \rightarrow q_k\) exists such that

1. \(q_0 = q_{\text{start}}\)
2. \(q_k = q_{\text{accept}}\)
3. for each \(i\), we have \(w_i \in L(R_i)\), where \(R_i = \delta(q_{i-1}, q_i)\) for each state \(q_i\) in the transition function.
• Let $k$ be the number of states of $G$
• If $k = 2$,
  - $G$ must consist of a start state, an accept state, and a single arrow connecting them with a regular expression $R$
  - Return $R$
• If $k > 2$,
  - Select any state $q_{rip}$ different from $q_{start}$ and $q_{accept}$
  - Let $G'$ be the GNFA $(Q', \Sigma, \delta', q_{start}, q_{accept})$
    + $Q' = Q - \{q_{rip}\}$
    + and for any $q_i \in Q' - \{q_{accept}\}$ and $q_j \in Q' - \{q_{start}\}$ let
      \[
      \delta'(q_i, q_j) = (R_1)(R_2)^* (R_3) \cup (R_4),
      \]
      where $R_1 = \delta(q_i, q_{rip})$,
      $R_2 = \delta(q_{rip}, q_{rip})$,
      $R_3 = \delta(q_{rip}, q_j)$, and
      $R_4 = \delta(q_i, q_j)$
  - Compute $\text{CONVERT}(G')$ and return this value

Proof

• Let $M$ be the DFA for language $A$
• Convert $M$ to a GNFA $G$ by adding a new start state and a single arrow
  - $G$ must consist of a start state, an accept state, and a single arrow
• If $l \geq 2$,
  - Let $l$ be the number of states of $G$
• Do $G$ and $G'$ accept the same language?

- Suppose $G$ accepts an input $w$.
  - There is an accepting sequence of states: $q_{\text{start}}, q_1, q_2, q_3, \ldots, q_{\text{accept}}$.
  - If none of them is $q_{\text{rip}}$, then $G'$ will accept $w$.

- Each of the new regular expressions is created from the old regular expressions.

- If none of them is $q_{\text{rip}}$, then $G'$ will accept $w$.

- If $q_{\text{rip}}$ does appear, remove each run of consecutive $q_{\text{rip}}$ states.

- This will be an accepting computation for $G'$.

- Each of the new regular expressions contains the old regular expression as part of a union.

- Induction on number of states $k$ in $G$.

Claim 1.65: For any G NFA, $\text{CONVERT}(G)$ is equivalent to $G$. 

Proof continued
Examples

• To understand the power of finite automata, you must understand their limitations.
  - Consider: 
    - \{w | w has an equal number of 0s and 1s and |w| is not regular\}
    - \{w | w has an equal number of 0s and 1s\}
    - \{w | w has an equal number of 0s and 1s\}
  - In the second class, we showed that \(a^n b^n\) is not regular.
  - What languages are not regular?
  - You must understand their limitations.
  - To understand the power of finite automata,

Overview

• Constructing Complex FA
  - Non-Regular Languages \(\Leftrightarrow\)
  - Regular Expressions (cont)
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 \( s \) can be divided into three parts, \( s = xyz \), satisfying the following conditions:

1. For each \( i \geq 0 \), \( xy^i z \in A \)
2. \( |y| > 0 \)
3. \( |xy| \leq p \)

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 \( s \) can be divided into three parts, \( s = xyz \), satisfying the following:

1. For each \( i \geq 0 \), \( xy^i z \in A \)
2. \( |y| > 0 \)
3. \( |xy| \leq p \)

If a string has length more than \( p \), then it must repeat a state, and everything that we saw between those two instances of that state could be taken out. This means that DFA must repeat a state, and so DFA can only be in one of those states. The language is recognizable by a DFA.

How can we prove a language is not regular?
Example:

\[
\{0^n 1^n \mid n \geq 0\}
\]

To use pumping lemma, assume that language is regular, and show there is a contradiction.

- Assume it is regular
  - Let \( p \) be the pumping length for the language
  - Choose \( s \) to be \( 0^p 1^p \)
  - By the pumping lemma, there must be a \( x, y, z \) in which \(|xy| \leq p\) such that \( y \neq \epsilon \) and for some \( j \geq 0 \),
    \[
    |x| + |y| = j
    \]
    \[
    |y| = k
    \]
    \[
    |z| \geq l
    \]
    where \( l \geq 0 \)
  - Choose \( s \) to be the pumping length for the language
  - Let \( d \) be the pumping length for the language
  - Assume \( s \) is regular

Show there is a contradiction.

Proof of Pumping Lemma

- Let \( M \) be a DFA that recognizes \( A \) and \( p \) be number of states
  - Use a DFA to prove this, much easier than using a NFA!
  - Case 1: There is no string in \( A \) of length at least \( d \)
    - Let \( N \) be a DFA that recognizes \( \{0^n 1^n \mid n \geq 0\} \)
    - By pigeonhole principle, two of the states must be the same, say \( q \).
    - In accepting \( s \), \( M \) will go through at least \( u + 1 \) states.
    - Case 2: Let \( s \) \( \in A \) and have length \( d \)
      - Let \( s = \epsilon \) and have length \( d \)

Proof of Pumping Lemma
Example: \( \{ \{ 0 \} \} \in \{ 0,1 \}^* \)

- This forces us to pump in the first set of 0's
- How about 010
- Won't work, as we can pump 00
- How about 010
- But what string should we choose?
- Assume it is regular, so must be a

\[ \{ \{ 0 \} \} \in \{ 0,1 \}^* \]

Example: \( \{ \{ 0 \} \} \in \{ 0,1 \}^* \)

- If we pick \((01)^d\), we cannot find contradiction
- How about \((0)^d\)?
- But what string should we choose?
- Assume it is regular, so must be a

\[ \{ \{ 0 \} \} \in \{ 0,1 \}^* \]
Overview

• Regular Expressions (cont)
• Non-regular languages
  ⇒ Constructing Complex FA

Final Words on Pumping Lemma

• How can you prove a language is regular?
  - See Question 1.54
  - Cannot use pumping lemma to show a language is regular
  - Some non-regular languages obey the pumping lemma
  - But not all non-regular languages obey the pumping lemma
  - All regular languages obey the pumping lemma

Final Words on Pumping Lemma
Epsilon Transitions

A is a regular language, and \( M \) a DFA that recognizes it.

Let \( L \) be \( \{ \epsilon \} \) if \( A \) is not empty, and \( \{ \} \) otherwise.

- We know that either way, \( L \) is a regular language.
- In either case, it is easy to build a DFA for \( L \).
- In fact, all finite languages are regular.

But let's say that we want to build a FA for \( L \) based on \( M \).

- Build a machine \( N \) for \( L \) based on \( M \), with same set of states.
- Change all transitions so that rather than read characters, they read \( \epsilon + \).

Note that \( M \) is now a NFA, but that is fine.

\[ \delta_N(q, \epsilon) \in \{ q' | \exists a \in \Sigma \text{ s.t. } \delta_M(q, a) = q' \} \]

Prefix DFA

\[ L = \{ w | wx \in A \} \]

where \( A \) is a regular language, \( \epsilon \) a regular language, and \( N \) a DFA that recognizes \( L \).

To construct a DFA for \( L \) use \( M \).

+ But change all of its non-garbage states to accept states.
+ Build a DFA \( N \) using \( M \).
+ Do not worry about states not reachable from start.
+ Since \( A \) is regular, there is a DFA \( N \) that recognizes it.

- Define garbage states:
  - garbage states:
    - Don't have to worry about states not reachable from start.
    - States that do not have a path to an accept state.
- Option 1:

  \[ \{ x \in \Sigma^* | \exists n \in \mathbb{N} \text{ s.t. } x = y^n \} \]
Second Way to Show Prefix is Regular

Let $A$ be a DFA for $L$ where $A \in \{a, b \mid \epsilon \} = L$

Let $M$ be a DFA for $A$

- Make a new machine $P$ that joins $M$ and $N$ together
- Have $\delta_N$'s transitions be same as $\delta_M$'s, but read $\epsilon$ instead of characters
- Make a copy of $N$, call it $N'$

Correctness of Proof

- If $A$ does not accept any string, there is no path from start to accept state in $N$, so $N'$ does not accept anything
- So there is no path in $N'$, so $N$ does not accept anything
Splicing

More formally:

Let \( M \) be a DFA s.t. \( L(M) = A \) and let \( N \) be a DFA s.t. \( L(N) = B \).

Join the two machines together:
- Add transitions from all states of \( M \) to all states of \( N \) with \( \varepsilon \) transition.
- Start state of \( P \) is \( M \)'s start state, and \( N \)'s accept states are the accept states of \( P \).

More formally:

- States of \( P \) are the union of states of \( M \) and \( N \).
- Start state of \( P \) is \( M \)'s start state, and accept states are \( N \)'s accept states.
- Transitions of \( P \) are the union of \( M \)'s transitions and \( N \)'s transitions.
- Accept states of \( P \) are \( N \)'s accept states.
- Transition \( \delta_P(q, \varepsilon) \rightarrow q' \) for all \( q \) in \( M \)'s states and \( q' \) in \( N \)'s states.

Now consider \( N \) similar to \( N \), same states, same start state, same accept states, \( \{ b \in \Sigma \, | \, b \notin (a \cdot b)^* \} = \emptyset \).

First, \( N \):

- \( \delta_P \) has all the following transitions:
  - \( \delta_P \) has all the transitions from all states of \( N \) to all states of \( N \) with \( \varepsilon \) transition.
  - \( \delta_P \) has all the transitions from all states of \( N \) to all states of \( N \) (not cross product).

---

\[ L(M) \cup L(N) \]
Common Prefix

Let $A$ be a DFA that recognizes $A$ and $N$ be a DFA that recognizes $B$.

Let $L = \{x \in A \land x \in B\}$.

Build machine $P$ in the same way we built a DFA that does intersection (cross product of states).

Turn all non-accepting states into accepting states.
Introduction

- Many languages can be described by FA and regular expressions
- Context-Free Grammars (CFGs) are more powerful
  - Can describe languages that have a recursive structure
  - Often used for human languages

- But some cannot

If you simplify this down to a FA, cannot capture what modifies what

\[ \text{NP} \rightarrow \text{N PP} \]
\[ \text{PP} \rightarrow \text{Prep NP} \]
\[ \text{NP} \rightarrow \text{N pp} \]

- Ambiguity
- Designing CFGs
- Context-Free Grammars

Overview
Context-Free Languages

- Any language that can be generated from a context-free grammar is a context-free language.
- Why is it called context-free grammar?

What is language of $G_1$?

- Can use $|$ to separate right-hand sides
- $L(G_1)$ is the language of $G_1$

Example $G_1$

$A \rightarrow 0A1$
$A \rightarrow B$
$B \rightarrow #$

All strings generated from grammar constitute the language of

Context-Free Languages

- Substitution rules (or production rules)
- Symbol (variable) arrow string (variables and terminals)
- Variables are usually upper case; terminals lower case
- Symbols (variables and terminals) are usually upper case (variables) or lower case (terminals)

• Can be represented as a parse tree
- Sequence of substitutions to obtain a string is called a derivation
- Replace a variable by right-hand side of one of its rules
- Repeated until no variables
- Write down the start symbol

Generating a string (computation)

- Symbol (variable) usually on left-hand side of first rule
- Variables are usually upper case; terminals lower case
- Symbols (variables) arrow string (variables and terminals)

Example $G_1$

$# \leftarrow B$
$A \leftarrow B$
$A \leftarrow V01$
Definition: A context-free grammar is a 4-tuple \((V, \Sigma, R, S)\), where

1. \(V\) is a finite set called the **variables**
2. \(\Sigma\) is a finite set, disjoint from \(V\), called the **terminals**
3. \(R\) is a finite set of **rules**, with each rule being a variable and a string of variables and terminals
4. \(S \in V\) is the **start variable**

where

\[
\text{Example 5: A context-free grammar is a 4-tuple \((V, \Sigma, R, S)\).}
\]
Example 69

\[ G_3 = ( \{ S \}, \{ a, b \}, R, S ) \]

where \( R = \{ S \to aSb | SS | \epsilon \} \)

**Computation**

**Definition:** If \( u, v, \) and \( w \) are strings of variables and terminals, and \( A \to w \) is a rule of the grammar, we say that \( uAv \) **yields** \( uwv \), written \( uAv \Rightarrow uwv \).

**Definition:** We say that \( u \) **derives** \( v \), written \( u \Rightarrow v \), if a sequence \( u = u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k = v \) exists for \( k \geq 0 \) and

\[ a = n \Rightarrow n \Rightarrow \ldots \Rightarrow n \Rightarrow b n \Rightarrow n \] and \( a \neq n \)

**Definition:** The **language of the grammar** is \( \{ w \in \Sigma^* | S \Rightarrow^* w \} \).

**Definition:** We say that \( n \) **derives** \( a \), written \( n \Rightarrow a \), if \( a \) is a rule of the grammar, we say that \( n \) **yields** \( a \).

**Definition:** If \( n, v \), and \( w \) are strings of variables and terminals,
Overview

• Context-Free Grammars
⇒ Designing CFGs
• Ambiguity

Example

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

$V = \{\langle \text{EXPR} \rangle, \langle \text{TERM} \rangle, \langle \text{FACTOR} \rangle\}$

$\Sigma = \{a, +, \times, (, )\}$

$R =$

$\langle \text{EXPR} \rangle \rightarrow \langle \text{EXPR} \rangle + \langle \text{TERM} \rangle | \langle \text{TERM} \rangle$

$\langle \text{TERM} \rangle \rightarrow \langle \text{TERM} \rangle \times \langle \text{FACTOR} \rangle | \langle \text{FACTOR} \rangle$

$\langle \text{FACTOR} \rangle \rightarrow (\langle \text{EXPR} \rangle) \mid a$

• How would we generate $a + a \times a$?

• How would we generate $(a + a) \times a$?

Parse captures the meaning a string

$\{i | i, +, \times, a, \} = \mathcal{L}$

$\{\langle \text{EXPR} \rangle, \langle \text{TERM} \rangle, \langle \text{FACTOR} \rangle\} = \mathcal{L}$

$G_4$
Designing: Hint 2

• If the language is regular - Construct a DFA for it - Convert DFA to a CFG:
  + Add a rule for each state $q_i$
  + Make a variable $R_i$ for each state $q_i$
  + Add the rule $R_i \rightarrow aR_j$ if $\delta(q_i, a) = q_j$
  + Or, if simpler, construct a NFA for it - Make a variable $R_i$ for each state $q_i$
  + Add the rule $R_i \rightarrow R_j$ if $q_j \in \delta(q_i, \epsilon)$

If the language is regular

Designing Context-Free Grammars

• Might be more difficult to program in than a FA, as we are not used to using grammars to describe problems
  • Hint 1: Many CFLs are the union of simpler CFLs
  - Construct a grammar for each part and then combine them with $S \rightarrow S_1 | S_2 | \ldots | S_k$ where $S_i$ is start variable for each individual grammar
  - Construct a grammar for each part and then combine them with $S \rightarrow S_1 \cup S_2 \cup \ldots \cup S_k$

Many CFLs are the union of simpler CFLs

Hint: Many CFLs are the union of simpler CFLs

Need to use grammars to describe problems more difficult to program in than a FA, as we are not
In more complex CFLs, strings may contain structures that appear recursively as parts of other structures.

Designing: Hint 4

- Place a variable that generates the structure in that location instead of an "a" that can occur alone. Parenthesized expression might appear multiple times.
- Example: arithmetic expression $a * b$ versus $p + c * (q + 6) * c + d$.
- Any place an 'a' can occur, an entire parenthesized expression might appear recursively as part of other structures.

Designing: Hint 3

- Some CFLs contain strings with two substrings that are linked.
- Machine needs to remember an unbounded amount of information about one of the substrings to verify that it corresponds properly to the other.
- Machine needs to remember an unbounded amount of information about one of the substrings with two substrings that are "linked."
Ambiguity

Sometimes a grammar can generate the same string in several ways.

- If a grammar derives a string in several ways, we say that the string is derivable ambiguously in the grammar.
- If at least one string is derivable ambiguously in the grammar, we say that the grammar is ambiguous.

But undesirable in a programming language.

+ Ambiguity in Natural Language: 'I saw Mary with the telescope.'
+ Undesirable in a programming language: 'a + b * c' doesn't always mean the same thing.

Intended meaning might be clear from context, intonation, world knowledge.

The distinction is important:

- Different ways
- Different meanings
- Different parses
- Different needs

Some examples of ambiguity in programming languages:

- 'a + b * c' doesn't always mean the same thing.
- 'a * b + c' doesn't always mean the same thing.
Another Example

Two parses for "the girl touches the boy with the flower"

- Is this bad?

\[
\begin{align*}
  \text{with} & \quad \left\langle \text{PREP} \right\rangle \\
  \text{touched} & \quad \left\langle \text{VERB} \right\rangle \\
  \text{boy} & \quad \left\langle \text{NOUN} \right\rangle \\
  \text{the} & \quad \left\langle \text{ARTICLE} \right\rangle \\
  \text{complex-Noun} & \quad \left\langle \text{CMPLX-NOUN} \right\rangle \\
  \text{complex-Verb} & \quad \left\langle \text{CMPLX-VERB} \right\rangle \\
  \text{Prep-Phrase} & \quad \left\langle \text{PREP-PHRASE} \right\rangle \\
  \text{Sentence} & \quad \left\langle \text{SENTENCE} \right\rangle \\
\end{align*}
\]

Example

What are the parses of \( a \times a + a \)?

Replace \( R \) with:

\[
\begin{align*}
  \left\langle \text{EXPR} \right\rangle & \quad \left\langle \text{EXPR} \right\rangle \\
  \left\langle \text{TERM} \right\rangle & \quad \left\langle \text{TERM} \right\rangle \times \left\langle \text{TERM} \right\rangle \\
  \left\langle \text{EXPR} \right\rangle & \quad \left\langle \text{EXPR} \right\rangle + \left\langle \text{EXPR} \right\rangle \\
  \left\langle \text{FACTOR} \right\rangle & \quad \left\langle \text{FACTOR} \right\rangle \\
\end{align*}
\]

\[
\begin{align*}
  \{ (\varnothing, \times, \cdot), a, +, \cdot \} & = \mathbb{A} \\
  \{ (\varnothing, \times, \cdot), a, \langle \text{EXPR} \rangle, \langle \text{TERM} \rangle \} & = \mathbb{G}_1 \\
\end{align*}
\]
Formal Definitions

• When we say a grammar generates a string ambiguously,
- It has two different parse trees, not two different derivations
+ Some derivations just different in what order rules are applied

• Sometimes, the language of an ambiguous grammar also has a non-ambiguous version (that generates the same language)
- Arithmetic has an unambiguous grammar that generates the same language, and so only assigns a single, meaningful, to each string

• Language is inherently ambiguous if no unambiguous grammar
- Example: \{y | \exists i,j,k \forall x \in \{a,b,c\} \text{ s.t. } y = x_1 i x_2 j x_3 k \}
- Ambiguity is if there are two different leftmost derivations
  - Can talk about leftmost derivation
  - At every step, replace leftmost variable
  - Some derivations just different in what order rules are applied
    + If has two different parse trees, not two different derivations

• When we say a grammar generates a string ambiguously,