Equivalence of regular expressions and regular languages (lukewarm leftover – from last week)

Non Regular Languages: Two Approaches
1. Myhill-Nerode Theorem
   (beautiful, but not in Sipser’s book)
2. the Pumping Lemma

Algorithmic questions for NFAs
Context Free Grammars (time permitting)

Sipser’s book, 1.4, 2.1, 2.2
Hopcroft and Ullman, 3.4
We now define generalized non-deterministic finite automata (GNFA).

An NFA:
- Each transition labeled with a symbol or $\varepsilon$,
- reads zero or one symbols,
- takes matching transition, if any.

A GNFA:
- Each transition labeled with a regular expression,
- reads zero or more symbols,
- takes transition whose regular expression matches string, if any.

GNFAs are natural generalization of NFAs.
A Special Form of GNFA

- **Start state** has outgoing arrows to every other state, but no incoming arrows.
- Unique **accept state** has incoming arrows from every other state, but no outgoing arrows.
- Except for start and accept states, arrows goes from every state to every other state, including itself.

Easy to transform any GNFA into special form.

Really? How? ... (saw this last week :-)
Converting DFA to Regular Expression (↔)

Strategy – sequence of equivalent transformations

- given a $k$-state DFA
- transform into $(k + 2)$-state GNFA
- while GNFA has more than 2 states, transform it into equivalent GNFA with one fewer state
- eventually reach 2-state GNFA (states are just start and accept).
- label on single transition is the desired regular expression.
Converting Strategy (↔)

- 3-state DFA
- 5-state GNFA
- 4-state GNFA
- 3-state GNFA
- 2-state GNFA

regular expression

Slides modified by Benny Chor, based on original slides by Maurice Herlihy, Brown University.
Removing One State

We remove one state $q_r$, and then repair the machine by altering regular expression of other transitions.

This is done with respect to any two edges going in and out of $q_r$. It results in getting rid of $q_r$, thus reducing number of states by one.
The CONVERT Algorithm

Given GNFA $G$, convert it to equivalent GNFA $G'$.

1. Let $k$ be the number of states of $G$.
2. If $k = 2$, return the regular expression labeling the only arrow.
3. If $k > 2$, select some $q_r (\neq q_s, q_a)$.
4. Let $Q' = Q - \{q_r\}$.
5. For any $q_i \in Q' - \{q_a\}$ and $q_j \in Q' - \{q_s\}$, let
   - $R_1 = \delta(q_i, q_r)$, $R_2 = \delta(q_r, q_r)$,
   - $R_3 = \delta(q_r, q_j)$, and $R_4 = \delta(q_i, q_j)$.
   - Notice that $i = j$ is also possible.
6. Define $\delta'(q_i, q_j) = (R_1)(R_2)^*(R_3) \cup (R_4)$.
7. Denote the resulting $k - 1$ states GNFA by $G'$. 

Slides modified by Benny Chor, based on original slides by Maurice Herlihy, Brown University.
The CONVERT Procedure

We define the recursive procedure \textsc{Convert}(\cdot):

Given GNFA $G$.

- Let $k$ be the number of states of $G$.
- If $k = 2$, return the regular expression labeling the only arrow of $G$.
- If $k > 2$, let $G'$ be the $k - 1$ states GNFA produced by the algorithm.

Return \textsc{Convert}($G'$).
Conversion - Example

- We now construct a simple, 2 state DFA that accepts the language over \( \{0, 1\} \) of all strings with an even number of 1s.

- We followed the conversion through GNFAs to translate this DFA (on the blackboard) into the regular expression \((0 \cup 10^*1)^*\).
Theorem: $G$ and $\text{CONVERT}(G)$ accept the same language.

Proof: By induction on number of states of $G$

Basis: When there are only 2 states, there is a single label, which characterizes the strings accepted by $G$.

Induction Step: Assume claim for $k - 1$ states, prove for $k$.

Let $G'$ be the $k - 1$ states GNFA produced from $G$ by the algorithm.
$G$ and $G'$ accept the same language

By the induction hypothesis, $G'$ and $\text{CONVERT}(G')$ accept the same language.

On input $G$, the procedure returns $\text{CONVERT}(G')$.

So to complete the proof, it suffices to show that $G$ and $G'$ accept the same language.

Three steps:

1. If $G$ accepts the string $w$, then so does $G'$.
2. If $G'$ accepts the string $w$, then so does $G$.
3. Therefore $G$ and $G'$ are equivalent.
Step One

**Claim:** If $G$ accepts $w$, then so does $G'$:

- If $G$ accepts $w$, then there exists a “path of states” $q_s, q_1, q_2, \ldots, q_a$ traversed by $G$ on $w$, leading to the accept state $q_a$.

- If $q_r$ does not appear on path, then $G'$ accepts $w$ because the new regular expression on each edge of $G'$ contains the old regular expression in the “union part”.

- If $q_r$ does appear, consider the regular expression corresponding to $\ldots q_i, q_r, \ldots, q_r, q_j \ldots$. The new regular expression $(R_{i,r})(R_{r,r})^*(R_{r,j})$ linking $q_i$ and $q_j$ encompasses any such string.

- In both cases, the claim holds.
Claim: If $G'$ accepts $w$, then so does $G$.

Proof: Each transition from $q_i$ to $q_j$ in $G'$ corresponds to a transition in $G$, either directly or through $q_r$. Thus if $G'$ accepts $w$, then so does $G$.

This completes the proof of the claim that $L(G) = L(G')$.

Combined with the induction hypothesis, this shows that $G$ and the regular expression $\text{CONVERT}(G)$ accept the same language.

This, in turn, proves our remarkable claim: A language, $L$, is described by a regular expression, $R$, if and only if $L$ is regular.
What We Just Completed

Thm.: A language, $L$, is described by a regular expression, $R$, if and only if $L$ is regular.

$\implies$ construct an NFA accepting $R$.

$\impliedby$ Given a regular language, $L$, construct an equivalent regular expression.
Negative Results

We have made quite some progress understanding what finite automata can do. But what can’t they do? Is there a DFA that accepts

- $B = \{0^n1^n|n \geq 0\}$
- $C = \{w|w$ has an equal number of 0’s and 1’s$\}$
- $D = \{w|w$ has an equal number of occurrences of 01 and 10 substrings$\}$

Consider $B$:

- DFA must “remember” how many 0’s it has seen
- impossible with finite state.

The others are exactly the same.
Negative Results

Is there a DFA that accepts

- \( B = \{0^n1^n | n \geq 0\} \)
- \( C = \{w | w \) has an equal number of 0’s and 1’s\} \)
- \( D = \{w | w \) is binary and has an equal number of occurrences of 01 and 10 substrings\} \)

Consider \( B \):

- DFA must “remember” how many 0’s it has seen
- impossible with finite state.

The others are exactly the same...

**Question:** This is sound intuition. But is this a proof?

**Answer:** No. \( D \) is regular!???

(related to assignment 1, prob. 5d, but not identical)
The Equivalence Relation $\sim_L$

Let $L \subseteq \Sigma^*$ be a language.

Define an equivalence relation $\sim_L$ on pairs of strings:

Let $x, y \in \Sigma^*$. We say that $x \sim_L y$ if for every string $z \in \Sigma^*$, $xz \in L$ if and only if $yz \in L$.

It is easy to see that $\sim_L$ is indeed an equivalence relation (reflexive, symmetric, transitive) on $\Sigma^*$.

In addition, if $x \sim_L y$ then for every string $z \in \Sigma^*$, $xz \sim_L yz$ as well (this is called right invariance).
The Equivalence Relation $\sim_L$

Like every equivalence relation, $\sim_L$ partitions $\Sigma^*$ to (disjoint) equivalence classes. For every string $x$, let $[x] \subseteq \Sigma^*$ denote its equivalence class w.r.t. $\sim_L$ (if $x \sim_L y$ then $[x] = [y]$ – equality of sets).

Question is, how many equivalence classes does $\sim_L$ induces?

In particular, is the number of equivalence classes of $\sim_L$ finite or infinite?

Well, it could be either finite or infinite. This depends on the language $L$. 
Classes of $\sim_L$: Three Examples

- Let $L_1 \subset \{0, 1\}^*$ contain all strings where the number of 1s is divisible by 4. Then $\sim_{L_1}$ has finitely many equivalence classes.

- Let $L_2 \subset \{0, 1\}^*$ contain all strings of the form $0^n1^n$. Then $\sim_{L_2}$ has infinitely many equivalence classes.

- Let $L_3 \subset \{1\}^*$ contain all strings whose length is a prime number. Then $\sim_{L_3}$ has infinitely many equivalence classes.

(white-board proofs.)
Myhill-Nerode Theorem

Theorem: Let $L \subseteq \Sigma^*$ be a language. Then $L$ is regular $\iff$ $\sim^L$ has finitely many equivalence classes.

Three specific consequences:

- $L_1 \subset \{0, 1\}^*$ contains all strings where the number of 1s is divisible by 4. Then $L_1$ is regular.

- $L_2 \subset \{0, 1\}^*$ contains all strings of the form $0^n1^n$. Then $L_2$ is not regular.

- Let $L_3 \subset \{1\}^*$ contains all strings whose length is a prime number. Then $L_3$ is not regular.
Suppose \( L \) is regular. Let \( M = (Q, \Sigma, \delta, q_0, F) \) be a DFA accepting it. For every \( x \in \Sigma^* \), let \( \delta(q_0, x) \in Q \) be the state where the computation of \( M \) on input \( x \) ends.

The relation \( \sim_M \) on pairs of strings is defined as follows:

\[ x \sim_M y \text{ if } \delta(q_0, x) = \delta(q_0, y). \]

Clearly, \( \sim_M \) is an equivalence relation.

Furthermore, if \( x \sim_M y \), then for every \( z \in \Sigma^* \), also \( xz \sim_M yz \). Therefore, \( xz \in L \) if and only if \( yz \in L \).

This means that \( x \sim_M y \implies x \sim_L y \).
Myhill-Nerode Theorem: Proof (cont.)

The equivalence relation $\sim_M$ has finitely many equivalence classes (at most the number of states in $M$).

We saw that $x \sim_M y \implies x \sim_L y$.

This means that the equivalence classes of $\sim_M$ refine those of $\sim_L$.

Thus, the number of equivalence classes of $\sim_M$ is greater or equal than the number of equivalence classes of $\sim_L$.

Therefore, $\sim_L$ has finitely many equivalence classes, as desired.
Suppose $\sim_L$ has finitely many equivalence classes. We’ll construct a DFA $M$ that accepts $L$.

Let $x_1, \ldots, x_n \in \Sigma^*$ be representatives for the finitely many equivalence classes of $\sim_L$.

The states of $M$ are the equivalence classes $[x_1], \ldots, [x_n]$.

The transition function $\delta$ is defined as follows: For all $a \in \Sigma$, $\delta([x_i], a) = [x_ia]$ (the equivalence class of $x_ia$).

Hey, why is this a proper definition of $\delta$? (Hint: right invariance).
The initial state is $[\varepsilon]$.

The accept states are $F = \{[x_i] \mid x_i \in L\}$.

Easy: On input $x \in \Sigma^*$, $M$ ends at state $[x]$ (why?).

Therefore $M$ accepts $x$ iff $x \in L$ (why?).

So $L$ is accepted by DFA, hence $L$ is regular.
Myhill-Nerode Theorem

- An example: Constructing DFA for the language $L_1 \subset \{0, 1\}^*$ contains all strings where the number of 1s is divisible by 5.

- A few words on minimizing the number of states of a DFA accepting a given language $L$. For example converting a four state DFA accepting $L = \{w \mid w \in \{0, 1\}^n \text{ and no. of ones is even}\}$ to an equivalent two state DFA.
We will show that all regular languages have a special property.

- Suppose $L$ is regular.

- If a string in $L$ is longer than a certain critical length $\ell$ (the pumping length),

- then it can be “pumped” to a longer string by repeating an internal substring any number of times.

- The longer string must be in $L$ too.

- This is a (second) powerful technique for showing that a language is not regular.
Pumping Lemma

Theorem: If $L$ is a regular language, then there is an $\ell > 0$ (the pumping length), where if $s$ is any string in $L$ of length $|s| > \ell$, then $s$ may be divided into three pieces $s = xyz$ such that

- for every $i \geq 0$, $xy^i z \in L$,
- $|y| > 0$, and
- $|xy| \leq \ell$.

Remarks: Without the second condition, the theorem would be trivial. The third condition is technical and sometimes useful.
Let \( M = (Q, \Sigma, \delta, q_1, F) \) be a DFA that accepts \( L \).

Let \( \ell = |Q| \), the number of states of \( M \).

If \( s \in L \) has length at least \( \ell \), consider the sequence of states \( M \) goes through as it reads \( s \):

\[
\begin{array}{cccccccc}
  s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & \ldots & s_n \\
  \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & & \uparrow \\
  q_1 & q_{20} & q_9 & q_{17} & q_{12} & q_{13} & q_9 & q_2 \\
\end{array}
\]

Since the sequence of states is of length \( |s| + 1 > \ell \), and there are only \( \ell \) different states in \( Q \), at least one state is repeated (by the pigeonhole principle).
Write down $s = xyz$

By inspection, $M$ accepts $xy^kz$ for every $k \geq 0$.

$|y| > 0$ because the state ($q_9$ in figure) is repeated.

To ensure that $|xy| \leq \ell$, pick first state repetition, which must occur no later than $\ell + 1$ states in sequence.
An Application

**Theorem:** The language $B = \{0^n1^n | n > 0\}$ is not regular.

**Proof:** By contradiction. Suppose $B$ is regular, accepted by DFA $M$. Let $\ell$ be the pumping length.

- Consider the string $s = 0^\ell 1^\ell$.
- By pumping lemma $s = xyz$, where $xy^kz \in B$ for every $k$.
- If $y$ is all 0, then $xy^kz$ has too many 0’s.
- If $y$ is all 1, then $xy^kz$ has too many 1’s.
- If $y$ is mixed, then $xy^kz$ is not of right form.

♣
Another Application

**Theorem**: The language 
\( C = \{ w \mid w \text{ has an equal number of 0's and 1's} \} \) is not regular.

**Proof**: By contradiction. Suppose \( C \) is regular, accepted by DFA \( M \). Let \( \ell \) be the pumping length.

- Consider the string \( s = 0^\ell 1^\ell \).
- By pumping lemma \( s = xyz \), where \( xy^kz \in C \) for every \( k \).
- If \( y \) is all 0, then \( xy^kz \) has too many 0’s.
- If \( y \) is all 1, then \( xy^kz \) has too many 1’s.
- If \( y \) is mixed, then since \( |xy| \leq \ell \), \( y \) must be all 0’s, contradiction.

♣
Theorem: The language $L_3 \subset \{1\}^*$, which contains all strings whose length is a prime number, is not regular.

Proof: By contradiction, using the pumping lemma and some simple properties of prime numbers.
Context Switch
Algorithmic Questions for NFAs

Q.: Given an NFA, \( N \), and a string \( s \), is \( s \in L(N) \)?

**Answer:** Construct the DFA equivalent to \( N \) and run it on \( w \).

Q.: Is \( L(N) = \emptyset \)?

**Answer:** This is a reachability question in graphs: Is there a path in the states’ graph of \( N \) from the start state to some accepting state. There are simple, efficient algorithms for this task.
More Algorithmic Questions for NFAs

Q.: Is \( L(N) = \Sigma^* \)?

Answer: Check if \( \overline{L(N)} = \emptyset \).

Q.: Given \( N_1 \) and \( N_2 \), is \( L(N_1) \subseteq L(N_2) \)?

Answer: Check if \( \overline{L(N_2)} \cap L(N_1) = \emptyset \).

Q.: Given \( N_1 \) and \( N_2 \), is \( L(N_1) = L(N_2) \)?

Answer: Check if \( L(N_1) \subseteq L(N_2) \) and \( L(N_2) \subseteq L(N_1) \).

In the future, we will see that for stronger models of computations, many of these problems cannot be solved by any algorithm.
Another, More Radical Context Switch

So far we saw
- finite automata,
- regular languages,
- regular expressions,
- Myhill-Nerode theorem and pumping lemma for regular languages.

We now introduce stronger machines and languages with more expressive power:
- pushdown automata,
- context-free languages,
- context-free grammars,
- pumping lemma for context-free languages.
Context-Free Grammars

An example of a context free grammar, $G_1$:

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

Terminology:

- Each line is a substitution rule or production.
- Each rule has the form: symbol $\rightarrow$ string. The left-hand symbol is a variable (usually upper-case).
- A string consists of variables and terminals.
- One variable is the start variable.
Rules for Generating Strings

- Write down the start variable (lhs of top rule).
- Pick a variable written down in current string and a derivation that starts with that variable.
- Replace that variable with right-hand side of that derivation.
- Repeat until no variables remain.
- Return final string (concatenation of terminals).

Process is inherently non deterministic.
Example

Grammar $G_1$:

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

Derivation with $G_1$:

\[
\begin{align*}
A & \Rightarrow 0A1 \\
& \Rightarrow 00A11 \\
& \Rightarrow 000A111 \\
& \Rightarrow 000B111 \\
& \Rightarrow 000\#111
\end{align*}
\]
A Parse Tree

Question: What strings can be generated in this way from the grammar $G_1$?
Answer: Exactly those of the form $0^n \#1^n \ (n \geq 0)$. 
Context-Free Languages

The language generated in this way is called the language of the grammar.

For example, $L(G_1)$ is $\{0^n \#1^n | n \geq 0\}$.

Any language generated by a context-free grammar is called a context-free language.
A Useful Abbreviation

Rules with same variable on left hand side

\[
\begin{align*}
A & \rightarrow 0A1 \\
A & \rightarrow B
\end{align*}
\]

are written as:

\[
A \rightarrow 0A1 \mid B
\]
A grammar $G_2$ to describe a few English sentences:

\[
\begin{align*}
\langle \text{SENTENCE} \rangle & \rightarrow \langle \text{NOUN-PHRASE} \rangle \langle \text{VERB} \rangle \\
\langle \text{NOUN-PHRASE} \rangle & \rightarrow \langle \text{ARTICLE} \rangle \langle \text{NOUN} \rangle \\
\langle \text{NOUN} \rangle & \rightarrow \text{boy} \mid \text{girl} \mid \text{flower} \\
\langle \text{ARTICLE} \rangle & \rightarrow \text{a} \mid \text{the} \\
\langle \text{VERB} \rangle & \rightarrow \text{touches} \mid \text{likes} \mid \text{sees}
\end{align*}
\]
Deriving English-like Sentences

A specific derivation in $G_2$:

\[
<\text{SENTENCE}> \Rightarrow <\text{NOUN-PHRASE} > <\text{VERB} > \\
\Rightarrow <\text{ARTICLE} > <\text{NOUN} > <\text{VERB} > \\
\Rightarrow a <\text{NOUN} > <\text{VERB} > \\
\Rightarrow a \text{ boy} <\text{VERB} > \\
\Rightarrow a \text{ boy sees}
\]

More strings generated by $G_2$:

a flower sees
the girl touches
Derivation and Parse Tree

\[
\begin{align*}
\langle \text{SENTENCE} \rangle & \Rightarrow \langle \text{NOUN-PHRASE} \rangle \langle \text{VERB} \rangle \\
& \Rightarrow \langle \text{ARTICLE} \rangle \langle \text{NOUN} \rangle \langle \text{VERB} \rangle \\
& \Rightarrow \text{a} \ \langle \text{NOUN} \rangle \langle \text{VERB} \rangle \\
& \Rightarrow \text{a boy} \ \langle \text{VERB} \rangle \\
& \Rightarrow \text{a boy sees}
\end{align*}
\]
Formal Definitions

A context-free grammar is a 4-tuple \((V, \Sigma, R, S)\) where

- \(V\) is a finite set of variables,
- \(\Sigma\) is a finite set of terminals,
- \(R\) is a finite set of rules: each rule is a variable and a finite string of variables and terminals.
- \(S\) is the start symbol.
Formal Definitions

If $u$ and $v$ are strings of variables and terminals, and $A \rightarrow w$ is a rule of the grammar, then we say $uAv$ yields $uwv$, written $uAv \Rightarrow uwv$.

We write $u \Rightarrow^* v$ if $u = v$ or

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

for some sequence $u_1, u_2, \ldots, u_k$.

Definition: The language of the grammar is

$$\left\{ w \in \Sigma^* \mid S \Rightarrow^* w \right\}.$$
Consider $G_4 = (V, \{a, b\}, R, S)$.

$R$ (Rules): $S \to aSb | SS | \varepsilon$.

Some words in the language: $aabb$, $aababb$.

Q.: But what is this language?

Hint: Think of parentheses.
Consider \((V, \Sigma, R, E)\) where

- \(V = \{E, T, F\}\)
- \(\Sigma = \{a, +, \times, (, )\}\)

Rules:

\[
\begin{align*}
    E & \rightarrow E + T | T \\
    T & \rightarrow T \times F | F \\
    F & \rightarrow (E) | a
\end{align*}
\]

Strings generated by the grammar:

- \(a + a \times a\) and \((a + a) \times a\).

What is the language of this grammar?

Hint: arithmetic expressions.

- \(E = \text{expression}\), \(T = \text{term}\), \(F = \text{factor}\).
Parse Tree for \( a + a \times a \)

\[
E \rightarrow E + T \mid T \\
T \rightarrow T \times F \mid F \\
F \rightarrow (E) \mid a
\]
Parse Tree for \((a + a) \times a\)

\[
\begin{align*}
E & \rightarrow E + T | T \\
T & \rightarrow T \times F | F \\
F & \rightarrow (E) | a
\end{align*}
\]
Designing Context-Free Grammars

No recipe in general, but few rules-of-thumb

- If CFG is the union of several CFGs, rename variables (not terminals) so they are disjoint, and add new rule $S \rightarrow S_1 \mid S_2 \mid \ldots \mid S_i$.

- To construct CFG for a regular language, “follow” a DFA for the language. For initial state $q_0$, make $R_0$ the start variable. For state transition $\delta(q_i, a) = q_j$ add rule $R_i \rightarrow aR_j$ to grammar. For each final state $q_f$, add rule $R_f \rightarrow \varepsilon$ to grammar.

- For languages with linked substrings (like $\{0^n \#1^n | n \geq 0\}$), a rule of form $R \rightarrow uRv$ may be helpful, forcing desired relation between substrings.
Closure Properties

- **Regular languages** are closed under
  - union
  - concatenation
  - star

- **Context-Free Languages** are closed under
  - union: $S \rightarrow S_1 | S_2$
  - concatenation $S \rightarrow S_1 S_2$
  - star $S \rightarrow \varepsilon | SS$
More Closure Properties

- Regular languages are also closed under
  - complement (reverse accept/non-accept states of DFA)
  - intersection \( L_1 \cap L_2 = \overline{L_1 \cup L_2} \).

- What about complement and intersection of context-free languages?
  - Not clear ...
Ambiguity

Grammar: \[ E \rightarrow E+E | E \times E | (E) | a \]
We say that a string $w$ is derived **ambiguously** from grammar $G$ if $w$ has two or more parse trees that generate it from $G$.

Ambiguity is usually not only a syntactic notion but also a **semantic** one, implying multiple meanings for the same string.

It is **sometime** possible to **eliminate** ambiguity by finding a different context free grammar generating the same language. This is true for the grammar above, which can be replaced by unambiguous grammar from slide 50.

Some languages (e.g. $\{1^i 2^j 3^k \mid i = j \text{ or } j = k\}$) are inherently ambiguous.
Chomsky Normal Form

A simplified, canonical form of context free grammars. Every rule has the form

\[
\begin{align*}
A & \rightarrow BC \\
A & \rightarrow a \\
S & \rightarrow \varepsilon
\end{align*}
\]

where $S$ is the start symbol, $A$, $B$ and $C$ are any variable, except $B$ and $C$ not the start symbol, and $A$ can be the start symbol.
Theorem: Any context-free language is generated by a context-free grammar in Chomsky normal form.

Basic idea:

- Add new start symbol $S_0$.
- Eliminate all $\varepsilon$ rules of the form $A \rightarrow \varepsilon$.
- Eliminate all “unit” rules of the form $A \rightarrow B$.
- Patch up rules so that grammar generates the same language.
- Convert remaining long rules to proper form.
Proof Idea

Add new start symbol $S_0$ and rule $S_0 \rightarrow S$. Guarantees that new start symbol does not appear on right-hand-side of a rule.
Proof

Eliminating $\varepsilon$ rules.

Repeat:

- remove some $A \rightarrow \varepsilon$.
- for each $R \rightarrow uAv$, add rule $R \rightarrow uv$.
- and so on: for $R \rightarrow uAvAw$ add $R \rightarrow uvAw$, $R \rightarrow uAvw$, and $R \rightarrow uvw$.
- for $R \rightarrow A$ add $R \rightarrow \varepsilon$, except if $R \rightarrow \varepsilon$ has already been removed.

until all $\varepsilon$-rules not involving the original start variable have been removed.
Proof

Eliminate unit rules.

Repeat:

- remove some $A \rightarrow B$.

- for each $B \rightarrow u$, add rule $A \rightarrow u$, unless this is previously removed unit rule. ($u$ is a string of variables and terminals.)

until all unit rules have been removed.
Proof

Finally, convert long rules.
To replace each $A \rightarrow u_1 u_2 \ldots u_k$ (for $k \geq 3$), introduce new non-terminals

$$N_1, N_2, \ldots, N_{k-1}$$

and rules

$$A \rightarrow u_1 N_1$$
$$N_1 \rightarrow u_2 N_2$$
$$\vdots$$
$$N_{k-3} \rightarrow u_{k-2} N_{k-2}$$
$$N_{k-2} \rightarrow u_{k-1} u_k$$
Conversion Example

Initial Grammar:

\[
\begin{align*}
S & \rightarrow ASA | aB \\
A & \rightarrow B | S \\
B & \rightarrow b | \varepsilon
\end{align*}
\]

(1) Add new start state:

\[
\begin{align*}
S_0 & \rightarrow S \\
S & \rightarrow ASA | aB \\
A & \rightarrow B | S \\
B & \rightarrow b | \varepsilon
\end{align*}
\]
Conversion Example (2)

\[
S_0 \rightarrow S \\
S \rightarrow ASA | aB \\
A \rightarrow B | S \\
B \rightarrow b | \varepsilon
\]

(2) Remove \(\varepsilon\)-rule \(B \rightarrow \varepsilon\):

\[
S_0 \rightarrow S \\
S' \rightarrow ASA | aB | a \\
A \rightarrow B | S | \varepsilon \\
B \rightarrow b | \varepsilon
\]
Conversion Example (3)

\[ S_0 \rightarrow S \]
\[ S \rightarrow ASA \mid aB \mid a \]
\[ A \rightarrow B \mid S \mid \varepsilon \]
\[ B \rightarrow b \]

(3) Remove \( \varepsilon \)-rule \( A \rightarrow \varepsilon \):

\[ S_0 \rightarrow S \]
\[ S \rightarrow ASA \mid aB \mid a \mid AS \mid SA \mid S \]
\[ A \rightarrow B \mid S \mid \varepsilon \]
\[ B \rightarrow b \]
Conversion Example (4)

\[
\begin{align*}
S_0 & \rightarrow S \\
S & \rightarrow ASA \mid aB \mid a \mid AS \mid SA \mid S \\
A & \rightarrow B \mid S \\
B & \rightarrow b
\end{align*}
\]

(4) Remove unit rule \( S \rightarrow S \)

\[
\begin{align*}
S_0 & \rightarrow S \\
S & \rightarrow ASA \mid aB \mid a \mid AS \mid SA \mid S \\
A & \rightarrow B \mid S \\
B & \rightarrow b
\end{align*}
\]
Conversion Example (5)

\[ S_0 \rightarrow S \]
\[ S \rightarrow ASA | aB | a | AS | SA \]
\[ A \rightarrow B | S \]
\[ B \rightarrow b \]

(5) Remove unit rule \( S_0 \rightarrow S \):

\[ S_0 \rightarrow S | ASA | aB | a | AS | SA \]
\[ S \rightarrow ASA | aB | a | AS | SA \]
\[ A \rightarrow B | S \]
\[ B \rightarrow b \]
Conversion Example (6)

\[
S_0 \rightarrow ASA \mid aB \mid a \mid AS \mid SA \\
S' \rightarrow ASA \mid aB \mid a \mid AS \mid SA \\
A \rightarrow B \mid S \\
B \rightarrow b
\]

(6) Remove unit rule \( A \rightarrow B \):

\[
S_0 \rightarrow ASA \mid aB \mid a \mid AS \mid SA \\
S' \rightarrow ASA \mid aB \mid a \mid AS \mid SA \\
A \rightarrow B \mid S \mid b \\
B \rightarrow b
\]
Conversion Example (7)

\[
\begin{align*}
S_0 & \rightarrow ASA | aB | a | AS | SA \\
S & \rightarrow ASA | aB | a | AS | SA \\
A & \rightarrow S | b \\
B & \rightarrow b
\end{align*}
\]

Remove unit rule \( A \rightarrow S \):

\[
\begin{align*}
S_0 & \rightarrow ASA | aB | a | AS | SA \\
S & \rightarrow ASA | aB | a | AS | SA \\
A & \rightarrow S | b | ASA | aB | a | AS | SA \\
B & \rightarrow b
\end{align*}
\]
Conversion Example (8)

\[
\begin{align*}
S_0 & \rightarrow ASA | aB | a | AS | SA \\
S & \rightarrow ASA | aB | a | AS | SA \\
A & \rightarrow b | ASA | aB | a | AS | SA \\
B & \rightarrow b
\end{align*}
\]

(8) Final simplification – treat long rules:

\[
\begin{align*}
S_0 & \rightarrow AA_1 | UB | a | SA | AS \\
S & \rightarrow AA_1 | UB | a | SA | AS \\
A & \rightarrow b | AA_1 | UB | a | SA | AS \\
A_1 & \rightarrow SA \\
U & \rightarrow a \\
B & \rightarrow b
\end{align*}
\]

√