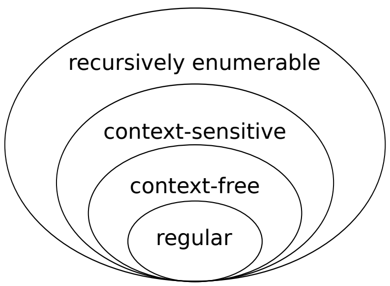
CS3100 Final Note Sheet

Notation

$\overline{L}$	Negation of language L.
L^R	The reversal of language L.
L^*	The Kleene-Star of L.
AB	The concatenation of language A and B.
$h(L)$	A homomorphism (a function that maps every input to a unique output) of L.
$A \setminus B$	Set difference of A and B. $A-B$ is the same thing.
2^A	The power-set of set A.

$f : x_1, x_2, \ldots, x_n \to y_1, y_2, \ldots, y_n$ denotes a function f that when given $x_1, \ldots, x_n$ as inputs yields $y_1, \ldots, y_n$ as outputs.

Chomsky Hierarchy



Regular Languages

A regular language is any language that can be recognized with a DFA. Formally a DFA is a tuple $(Q, \Sigma, \delta, q_0, F)$. Where:

Q	A finite, non-empty set of states.
Σ	A finite, non-empty alphabet.
δ	A function ($\delta : Q \times \Sigma \to Q$) that maps a state, and an input in Σ to a new state.
q_0	A state in Q that DFA starts execution from.
$F \subseteq Q$	A finite, possibly empty, set of accepting states.

Alternatively, regular languages can be defined by an NFA. Formally, NFAs are the same as DFAs, except the δ function for NFAs is defined as:

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \ \to \ 2^Q$$

Where 2^Q represents the power-set of Q. Basically, the delta function can now map an input to multiple states instead of just one state.

Closures

Where R is a regular language, L is 'not regular', and ? is Unknown.

Closed:		**Unclosed:**
$\overline{R} \to R$	$h(R) \to R$	$R \cap L \to ?$
$R^* \to R$	$R \cup R \to R$	$R \cup L \to ?$
$R^R \to R$	$R \cap R \to R$	$L \cup L \to ?$
$RR \to R$	$R \setminus R \to R$	

Pumping Lemma

$$\begin{aligned}
&\exists N \in \mathbb{N} : \\
&\quad \forall w \in L : |w| \geq N \Rightarrow \\
&\exists xyz \in \Sigma^* : \quad w = xyz \\
&\qquad \wedge |xy| \leq N \\
&\qquad \wedge y \neq \varepsilon \\
&\qquad \wedge \forall i \geq 0 : xy^iz \in L
\end{aligned}$$

Context Free Languages

Closures

Where C is a context-free language, R is a regular language, and ? is an Unknown language.

Closed:		**Unclosed:**
$C^R \to C$	$C \cup C \to C$	$\overline{C} \to ?$
$C^* \to C$	$C \cap R \to C$	$C \cap C \to ?$
$CC \to C$	$C \cup R \to R$	$C \setminus C \to ?$
$h(C) \to C$		

Pumping Lemma

$$\begin{aligned}
&\exists N \in \mathbb{N} : \\
&\quad \forall w \in L : |w| \geq N \Rightarrow \\
&\exists uvxyz \in \Sigma^* : \quad w = uvxyz \\
&\qquad \wedge |vy| > 0 \\
&\qquad \wedge |vxy| \leq N \\
&\qquad \wedge \forall i \geq 0 : uv^ixy^iz \in L
\end{aligned}$$

Ambiguous Context Free Languages

These are languages that have two separate parse-trees. To prove that a language is ambiguous, show that it actually has two separate parse-trees.

Example ambiguous grammar:

$$E \ \to \ E + E \mid E * E \mid \texttt{NUMBER}$$

Consistency and Completeness

Consistency: All strings generated by a grammar are in the language.

Completeness: The grammar generates all strings in the language.

You cannot know that a grammar defines a language until you show both. For example, if we want to define the language $\{a^nb^n | n \in \mathbb{N}\}$, the grammar:

$$S \ \to \ aabb$$

Is *consistent* because it only generates strings in the language, but not complete because it doesn't generate all strings in the language. Likewise, the grammar:

$$S \ \to \ aS \mid bS \mid \varepsilon \qquad (\textit{grammar for } \{a, b\}^*)$$

Is complete, it generates all possible strings in the language, but not consistent because it generates many strings that are, in-fact, outside of the language.

Cardinality

Diagonalization

Schröder-Bernstein Theorem

The Schröder-Bernstein theorem states that, for any two sets A and B if there exists an *injective* function from $A \to B$, and there exits an injective function from $B \to A$, then $|A| = |B|$. Note that the injective function doesn't require every item of A to map to every item of B, only that every item of A maps to *an* item of B (and vice versa).

Pairing Functions

Turing Machines

Terminology and Notation

$\langle M \rangle$	String representation of Turing machine M.
halting	When a machine stops execution.
acceptance	When a machine halts in a final state.
rejection	When a machine halts and is not in a final state.
decider	A decider is a Turing machine that defines a language of Turing machines that conform to a yes or no question.

Post's Correspondence Problem

Decidability

Halting Problem

The halting problem states building a Turing machine P that can detect whether any other Turing machine will halt is impossible. The proof is as follows:

Assume that we have a Turing machine P, that when given a Turing machine M and string w as input $(\langle M, w \rangle)$, P will (in a finite computation time) accept in the case that M halts on input w, or reject in the case that M loops on input w. We can then define a new Turing machine Q that takes a single Turing machine M as input. Q will then ask P whether machine M halts when given itself as input (does $P(\langle M, M \rangle)$ halt?). If P accepts (says that M halts) the Q will loop. If P rejects (says M will loop) then Q halts. Now, we can supply Q as input to machine Q. Q will then run $P(\langle Q, Q \rangle)$. If P accepts, then Q will begin to loop, but P said that Q would halt. This is a contraction, a general P decider for the halting problem cannot exist.

Mapping Reduction

A mapping reduction between $A \subseteq \Sigma^*$ and $B \subseteq \Sigma^*$ is a function $f : \Sigma^* \to \Sigma^*$ if $\forall x \in \Sigma^*, x \in A \Leftrightarrow f(x) \in B$. More plainly, a function f such that I can pick any x in A, and $f(x)$ will also be in B. The "mapping reduction from A to B" is typically denoted as $A \leq_m B$. A mapping reduction in polynomial time is denoted $A \leq_p B$.

The general steps for a mapping reduction $A \leq_m B$ are as follows:

1. A is designated the "known undecidable" language.
2. B is designated the "unknown" language.
3. Create a function f that maps all elements of A into B.

To form f you usually assume the decider for B (D_B), then you construct a machine M that uses to decider D_B to become a decider for A (D_A). For example, we can map A_{TM} onto $Halt_{TM}$ using the following method:

Assume that a decider R for A_{TM} exists. We will now construct a decider S for $Halt_{TM}$ from R. S has two inputs a machine M and an input string w. First, S will run decider R on $\langle M, w$, if R accepts, then S accepts. If R rejects, then S accepts. We now have a decider for $Halt_{TM}$ which is undecidable, a decider for A_{TM} cannot exist.

Rice's Theorem

"Every non-trivial partitioning of the space of Turing machine codes based on the languages recognized by these Turing machines is undecidable."

More formally, given a property $\mathcal{P}$, where $\mathcal{P}$ is non-trivial (not $\emptyset$ or Σ^*) the language below is undecidable.

$$\langle M \rangle \mid M \text{ is a Turing machine and } \mathcal{P}(Lang(M))$$

NP-Completeness

Problem Classes

P: The set of problems that can be solved in polynomial time. Contained in NP.

NP: The set of problems that can be solved in non-deterministic polynomial time.

NP-hard: The set of problems that can be polynomial time reduced to every other problem in NP.

NP-complete: The set of problems that are in both NP, and NP-hard.

Proving NP-Completeness

There are two steps to proving that a languages is in NP-complete. First you have to show that it is NP, and then you have to show that it is in NP-hard.

Verifiers

One way to show that a problem is in NP is by using a verifier. A verifier is a Turing machine V_L such that for all $w \in \Sigma^*$, their exists some c such that $w \in L$ when $V_L(w, c)$ accepts. Intuitively this can be understood as "There is a machine that can check the answers to problems quickly".

NP-hard Reduction