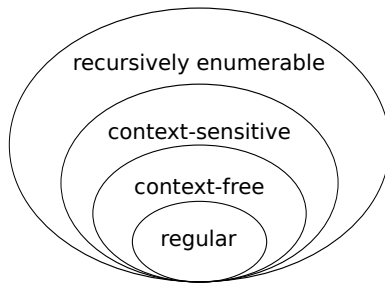


CS3100 Final Note Sheet

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 .
$f : x_1, x_2, \dots, x_n \rightarrow y_1, y_2, \dots, y_n$	denotes a function f that when given x_1, \dots, x_n as inputs yields y_1, \dots, 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 ($\delta : Q \times \Sigma \rightarrow Q$) 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:

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

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:

$$\begin{aligned} \overline{R} &\rightarrow R & h(R) &\rightarrow R \\ R^* &\rightarrow R & R \cup R &\rightarrow R \\ R^R &\rightarrow R & R \cap R &\rightarrow R \\ RR &\rightarrow R & R \setminus R &\rightarrow R \end{aligned}$$

Unclosed:

$$\begin{aligned} R \cap L &\rightarrow ? \\ R \cup L &\rightarrow ? \\ L \cup L &\rightarrow ? \end{aligned}$$

Pumping Lemma

$$\exists N \in \mathbb{N} :$$

$$\forall w \in L : |w| \geq N \Rightarrow$$

$$\exists xyz \in \Sigma^* : w = xyz$$

$$\wedge |xy| \leq N$$

$$\wedge y \neq \varepsilon$$

$$\wedge \forall i \geq 0 : xy^i z \in L$$

Operations

DFA Negation: Mark all non-final states final and all final states non-final.

DFA Reversal: Introduce a new initial state ' q_I '. Add ε transitions from this new state to all old final states and mark these states as non-final. Reverse all arrows in the DFA, and then mark the old initial state q_0 as final.

DFA Concatenating (AB): Add ε transitions from all of A 's final states to B 's initial state. Mark A 's final states as non-final.

DFA Union ($A \cup B$): Set the Q parameter of the new DFA to A 's Q crossed with B 's Q : $Q_{\text{new}} = A_Q \times B_Q$, Q will now contain pairs. Now change the δ function to be in terms of both A 's delta function and B 's delta function: $\delta_{\text{new}} = f((q_A, q_B), s) \rightarrow (\delta_A(q_A, s), \delta_B(q_B, s))$. That is to say, if A had state $q0_A$ that went to state $q1_A$ on input 0, and B had state $q2_B$ that went to state $q3_B$ state on input 0, then the new state $(q0_A, q2_B)$ would go to state $(q1_A, q3_B)$ on input 0.

The state that contains both DFA's initial state becomes the new initial state. New states where **either** item in the pair were final states, become final.

DFA Intersection ($A \cap B$): Exactly the same as DFA union except only pairs of states where **both** states in the pair are final become final states.

Conversions

NFA to DFA

Assume a function $\varepsilon\text{-closure}(x)$ that when given a set of states from the NFA as input returns the set of states that can be reached from these states via ε transition.

1. The initial state of the new DFA is $\varepsilon\text{-closure}(\{q_0\})$, where q_0 is the initial state of the NFA.
2. Next, form a table where the rows are the states in the DFA (to start only write the initial state) and the columns are characters in Σ . Now proceed down rows filling in the columns for each state using the following equation. If the current state is S and the column's symbol is i , the the cell S, i is:

$$\delta(S, i) = \varepsilon\text{-closure}\left(\bigcup_{s \in S} \delta(s, i)\right)$$

In English, the $\varepsilon\text{-closure}$ of the union of all states that can be reached from this DFA state's component NFA states on input i .

If the output state of the above function is not yet in the table, add it as a new row and process it as normal.

3. Once this process is complete, you should be able to convert the table into a standard graph by using the columns to make the transitions. **States where any component state is final, are final.**

RE to NFA

DFA to RE

DFA Minimization

Table Conversion Algorithm

Brzowski's Algorithm

This algorithm is quite simple. The crux of it is that an NFA to DFA conversion naturally results in a minimization of the NFA. So the algorithm is as-follows: Take

a DFA, reverse it to get an NFA, convert that NFA to a DFA again, take this new reversed DFA and reverse it to get the original language back, then convert the NFA resulting from the reversal into a DFA.

Context Free Languages

Closures

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

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

Pumping Lemma

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

CFG to PDA Conversion

Chomsky Normal Form

Cocke-Kasami-Younger (CKY) Parsing

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:

$$E \rightarrow E + E \mid E * E \mid \text{NUMBER}$$

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 $\{a^n b^n \mid n \in \mathbb{N}\}$, the grammar:

$$S \rightarrow aabb$$

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:

$$S \rightarrow aS \mid bS \mid \varepsilon \quad (\text{grammar for } \{a, b\}^*)$$

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 $A \rightarrow B$, and there exists an injective function from $B \rightarrow A$, 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

Turing Recognizable & Recursively Enumerable

Bang.

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 contradiction, 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 $f : \Sigma^* \rightarrow \Sigma^*$ 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 $\langle M, w \rangle$, 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 language 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 $w \in \Sigma^*$, there 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”.

[NEEDS MORE]

Deciders

NP-hard Reduction