CSC B36 — Introduction to the Theory of Computation

Compiled by Ali Rajan

Fall 2023

This is a compilation of the notes from Professor Nick Cheng's CSC B36 lectures. The facts (definitions, theorems, etc.) and examples are numbered for cross-referencing purposes. For examples from *Introduction to the Theory of Computation — Course notes for CSC B36/236/240*, the corresponding example numbers in the notes are provided.

Contents

Week 1 Logic and Proofs

The lectures in this week covered the first three additional notes, namely:

- What you should know about proofs
- How NOT to write proofs
- Common ways to write statements in English

Week 2 Induction

2.1 Principle of Simple Induction (PSI)

Let $b \in \mathbb{N}$ and P be a predicate on $\mathbb{N}$. If we prove the

- Basis step: $P(b)$
- Induction step: for all $i \geq b$, if $P(i)$ holds, then $P(i+1)$ holds

then we can conclude that for all $n \geq b$, $P(n)$. This is because

- $P(b)$ is true from the basis
- $P(b+1)$ holds from one application of the induction step
- $P(b+2)$ holds from two applications of the induction step

and so on.

Example 2.1 (Stamps — Example 1.7)
Let $P(n)$ be the predicate that exact postage of n cents can be made using only 4 cent and 7 cent stamps. Prove that $\forall n \in \mathbb{N}, P(n)$.

$P(n)$ can be equivalently stated as $Q(n)$: there exist $k, l \in \mathbb{N}$ such that $4k + 7l = n$.

Basis: For $n = 18$, let $k = 1$ and $l = 2$. It follows that $4k + 7l = 18$, as desired.

Induction Step: Let $n \geq 18$. Suppose that there exist $k, l \in \mathbb{N}$ such that $4k + 7l = n$ (I.H.).

We want to show that there exist $k', l' \in \mathbb{N}$ such that $4k' + 7l' = n + 1$. Consider two cases: $l > 0$ and $l = 0$.

For $l > 0$, let $k' = k + 2$ and $l' = l - 1$. We have $k', l' \in \mathbb{N}$ and

$$\begin{aligned} 4k' + 7l' &= 4(k+2) + 7(l-1) \\ &= 4k + 7l + 8 - 7 \\ &= n + 1 && \text{(by the I.H.)} \end{aligned}$$

as desired.

For $l = 0$, we have $4k = n \geq 18$, so $k \geq 5$. Let $k' = k - 5$ and $l' = l + 3$. Thus, $k', l' \in \mathbb{N}$ and

$$\begin{aligned} 4k' + 7l' &= 4(k-5) + 7(l+3) \\ &= 4k + 7l - 20 + 21 \\ &= n + 1 && \text{(by the I.H.)} \end{aligned}$$

as desired.

Remark: *The inductive hypothesis should be clearly labelled (e.g. using I.H.) and be stated in the induction step (not separately, e.g. under a separate inductive hypothesis header).*

2.2 Principle of Complete Induction (PCI)

Let $b \in \mathbb{N}$ (the first base case) and P be a predicate on $\mathbb{N}$ and $k \in \mathbb{N}$ (the number of base cases). If we prove the:

- Basis step: $P(b)$, $P(b+1)$, and $P(b+k-1)$
- Induction step: For all $i \geq b + k$, if $P(j)$ holds whenever $b \leq j < i$, then $P(i)$ holds

then we can conclude that for all $n \geq b$, $P(n)$.

Example 2.2 (Stamps — Example 1.12)
Let $Q(n)$ be the predicate that exact postage of n cents can be made using only 4 cent and 7 cent stamps. That is, there exist $k, l \in \mathbb{N}$ such that $4k + 7l = n$. Use the principle of complete induction to prove that $\forall n \geq 18$, $Q(n)$.

Basis Step: Consider four cases: $n \in \{18, 19, 20, 21\}$.

For $n = 18$, let $k = 1$ and $l = 2$. We have $4k + 7l = 18 = n$, as desired.

For $n = 19$, let $k = 3$ and $l = 1$. We have $4k + 7l = 19 = n$, as desired.

For $n = 20$, let $k = 5$ and $l = 0$. We have $4k + 7l = 20 = n$, as desired.

For $n = 21$, let $k = 0$ and $l = 3$. We have $4k + 7l = 21 = n$, as desired.

INDUCTION STEP: Let $n \geq 22$. Suppose that $P(j)$ holds whenever $18 \leq j < n$ (I.H.). We want to show that $Q(n)$ holds.

Since $n \geq 22$, $18 \leq n - 4 < 22$, so $Q(n-4)$ holds. Thus, there exist $k, l \in \mathbb{N}$ such that $4k + 7l = n - 4$. Let $k' = k + 1$ and $l' = l$. We have $k', l' \in \mathbb{N}$ and

$$\begin{aligned} 4k' + 7l' &= 4(k+1) + 7l \\ &= 4k + 7l + 4 \\ &= (n-4) + 4 && \text{(by the I.H.)} \\ &= n \end{aligned}$$

as desired.

Remark: *Whenever considering separate cases in a proof, they should be clearly stated as the basis step in example 2.2 does.*

2.3 Recurrences

Remark: *Recurrences will not be tested in this course.*

Definition 2.3 (Recurrence)
Recurrences are inductively/recursively defined functions.

Example 2.4 (Fibonacci Numbers)
One recurrence is

$$\text{Fib}(n) = \begin{cases} n & \text{if } 0 \leq m \leq 1 \\ \text{Fib}(n-1) + \text{Fib}(n-2) & \text{if } n > 1 \end{cases}$$

Example 2.5 (Unwinding a Recurrence)
Let $C(n)$ be the number of data assignments in sorting a list of n items using merge sort. We have

$$C(n) = \begin{cases} 0 & \text{if } n = 1 \\ C\left(\lfloor \frac{n}{2} \rfloor\right) + C\left(\lceil \frac{n}{2} \rceil\right) & \text{if } n > 1 \end{cases}$$

Let n be a "large" power of 2. That is, $n = 2^k$ for some "large" $k \in \mathbb{N}$. We get

$$\begin{aligned} C(n) &= 2C\left(\frac{n}{2}\right) + 2n && (1^{\text{st}} \text{ iteration}) \\ &= 2\left(2C\left(\frac{n}{4}\right) + 2 \cdot \frac{n}{2}\right) + 2n && (2^{\text{nd}} \text{ iteration}) \\ &= 2^2 C\left(\frac{n}{4}\right) + 4n \\ &= 2^2\left(2C\left(\frac{n}{8}\right) + 2 \cdot \frac{n}{4}\right) + 4n && (3^{\text{rd}} \text{ iteration}) \\ &= 2^3 C\left(\frac{n}{8}\right) + 6n \\ &\vdots \\ &= 2^i C\left(\frac{n}{2^i}\right) + 2in && (i^{\text{th}} \text{ iteration}) \\ &\vdots \\ &= 2^k C\left(\frac{n}{2^k}\right) + 2kn && (k^{\text{th}} \text{ iteration}) \\ &= nC(1) + 2(\log_2 n)n \\ &= 2n \log_2 n \end{aligned}$$

2.4 Structural Induction

There are two uses of structural induction:

1. Defining sets
2. Proving properties about the elements in a structural induction defined set

Example 2.6 (Well-Formed Algebraic Expressions — Example 4.1)
Consider algebraic expressions with variables x, y, and z, and binary operators $+$, $-$, $\times$, and $\div$. More concretely, the expressions are strings with characters from $\{x, y, z, +, -, \times, \div, (,)\}$.

Definition 2.7 (Minimal Set of Well-Formed Algebraic Expressions)
Let $\mathcal{E}$ be the smallest set such that

Basis: $x, y, z \in \mathcal{E}$.

Induction Step: If $e_1, e_2 \in \mathcal{E}$, then $(e_1 + e_2), (e_1 - e_2), (e_1 \times e_2), (e_1 \div e_2) \in \mathcal{E}$.

Example 2.8 (Structural Induction Proof — Example 4.2)
Let $\mathcal{E}$ be as defined earlier in definition 2.7. For a string $e \in \{x, y, z, +, -, \times, \div, (,)\}^*$, define

- $\mathbf{vr}(e)$ as the number of occurrences of variables in e
- $\mathbf{op}(e)$ as the number of occurrences of operators in e

For example, $\mathbf{vr}\big(((x + y) \times z)\big) = 3$, $\mathbf{op}\big(((x + y) \times z)\big) = 2$, $\mathbf{vr}\big((x + x)\big) = 2$, and $\mathbf{op}\big(((x + y) + z)\big) = 2$.

Define a predicate (on $\mathcal{E}$) $P(e)$: $\mathbf{vr}(e) + \mathbf{op}(e) + 1$. Prove that $\forall e \in \mathcal{E}, P(e)$. Use a structural induction proof.

Proof: We proceed using induction on e.

Basis: Consider three cases: $e = x$, $e = y$, and $e = z$.

For $e = x$, $\mathbf{vr}(e) = 1$ and $\mathbf{op}(e) = 0$, so $\mathbf{vr}(e) = \mathbf{op}(e) + 1$, as desired. By similar reasoning, the claim also holds for $e = y$ and $e = z$.

Induction Step: Let $e_1, e_2 \in \mathcal{E}$. Suppose that $P(e_1)$ and $P(e_2)$ hold; that is, $\mathbf{vr}(e_1) = \mathbf{op}(e_1) + 1$ and $\mathbf{vr}(e_2) = \mathbf{op}(e_2) + 1$ (I.H.). Consider the four cases $e = (e_1 + e_2)$, $e = (e_1 - e_2)$, $e = (e_1 \times e_2)$, and $e = (e_1 \div e_2)$.

For $e = (e_1 + e_2)$, $\mathbf{vr}(e) = \mathbf{vr}(e_1) + \mathbf{vr}(e_2)$ and $\mathbf{op}(e) = \mathbf{op}(e_1) + \mathbf{op}(e_2) + 1$. Thus,

$$\begin{aligned} \mathbf{vr}(e) &= \mathbf{vr}(e_1) + \mathbf{vr}(e_2) \\ &= (\mathbf{op}(e_1) + 1) + (\mathbf{op}(e_2) + 1) && \text{(by the I.H.)} \\ &= \mathbf{op}(e) + 1 \end{aligned}$$

By similar reasoning, the claim holds for the other three cases as well. ∎

Example 2.9
Define S to be the *smallest set* such that

Basis: $0 \in S$

Induction Step: If $n \in S$, then $n + 1 \in S$.

We have $S = \mathbb{N}$. Note that $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$ satisfy all the previous conditions except for being the smallest such set.

Week 3 Program Correctness

The lectures in this week covered the seventh to ninth additional notes (inclusive), namely:

- Pseudocode used for program correctness
- Proving program correctness
- Sample correctness proofs (this was partially covered, and is continued in week 4)

The remaining notes for this week contain an extra example and some information on acceptable ways to write predicates for recursive program correctness proofs.

3.1 Correctness of Iterative Programs

Example 3.1
Consider the following algorithm.
Precondition: A is an array of integers.
Postcondition: A is sorted.

```
1 SORT(A)
2     for i = 0 to len(A) − 1:
3         A[i] = i
```

For the specified postcondition, the algorithm is correct, even though it does not sort the values in the array.

3.2 Correctness of Recursive Programs

For the recursive example in the "Sample correctness proofs" additional notes, the predicate can instead be defined as

$Q(n)$: If $n \in \mathbb{N}$, then $\text{SQ}(n)$ returns n^2.

Note that the input size n in the example was the input itself, so a separate variable did not have to be defined for the input size. Additionally, stating that the program "returns" a value rather than "*terminates* and returns" the value is sufficient.

Week 4 Program Correctness Continued

4.1 Correctness of Binary Search

Example 4.1 (Correctness of Iterative Binary Search)
We will prove that the iterative version of binary search on the course website is correct. The program is as follows:

```
Pre:  A is a nonempty, sorted list of integers,
      x is an integer.
Post: Return True if x is in A;
      otherwise return False.

BSearchIter(A, x):
1. lo = 0;   hi = len(A)
2. while lo+1 < hi:
3.    mid = floor( (lo+hi) / 2 )
4.    if x < A[mid]:   hi = mid
5.    else:   lo = mid
6. if A[lo] == x:   return True
7. else:   return False
```

Proof:

Loop Invariant (LI):

(a) $0 \le lo < hi \le \text{len}(A)$

(b) If $x \in A$, then $x \in A[lo\colon hi]$.

Basis: On entry to the loop, $lo = 0$ and $hi = \text{len}(A) > 0$ (as A is non-empty). Therefore, $0 \le lo < hi \le \text{len}(A)$, as desired for LI (a).

Also, $A[lo\colon hi] = A\big[0\colon \text{len}(A)\big] = A$, so LI (b) follows.

Induction Step: Consider an arbitrary iteration. Suppose the LI holds before the iteration (IH). We want to prove that the LI holds after the iteration.

By line 3, $mid = \left\lfloor \frac{lo+hi}{2} \right\rfloor$. By PL2.2, $lo < mid < hi$. Consider two cases: $x < A[mid]$ and $x \ge A[mid]$.

If $x < A[mid]$, then $lo' = lo$ and $hi' = mid$ (line 4). Thus,

$$0 \underbrace{\le}_{\text{LI (a)}} lo' \underbrace{=}_{\text{line 4}} lo \underbrace{<}_{\text{PL2.2}} mid \underbrace{=}_{\text{line 4}} hi' \underbrace{<}_{\text{PL2.2}} hi \underbrace{\le}_{\text{LI (a)}} \text{len}(A)$$

as desired for LI (a).

If $x \in A$, then $x \in A[lo\colon hi]$ (by LI (b)) since $x < A[mid] \implies x \notin A[mid\colon hi]$ as $x \in A[lo\colon mid]$. This proves LI (b).

Insert $x \ge A[mid]$ case here.

Partial Correctness: Suppose the loop terminates. Consider the values of lo and hi on exit.

By the exit condition, $lo + 1 \ge hi$. By LI (a), $lo < hi$ (as $lo + 1 \le hi$). Thus, $lo + 1 = hi$, so $A[lo]$ is the only element in $A[lo\colon hi]$.

If $x \in A$, then $x \in A[lo\colon hi] = A[lo]$. By line 6, True is returned, as desired. If $x \notin A$, then $x \ne A[lo]$. By lines 6–7, False is returned, as desired.

Termination: For each iteration, we associate the expression $e = hi - lo$.

By LI (a), $lo < hi$, so $e > 0$. Consider an arbitrary iteration. We have

$$\begin{aligned} e' &= hi' - lo' \\ &= \begin{cases} mid - lo & \text{if } x < A[mid] \\ hi - mid & \text{if } x \ge A[mid] \end{cases} && \text{(where } mid = \left\lfloor \frac{lo+hi}{2} \right\rfloor \text{, by lines 4–5)} \end{aligned}$$

$$= \begin{cases} hi' - lo' \\ hi' - lo' \end{cases} \qquad (lo < mid < hi, \text{ as shown in the induction step for LI})$$

$$< hi - lo \qquad (\text{by PL2.2})$$

as desired. ■

Example 4.2 (Correctness of Recursive Binary Search)
We will prove that the recursive version of binary search on the course website is correct. The program is as follows:

```
Pre:  A is a nonempty, sorted list of integers,
      x is an integer.
Post: Return True if x is in A;
      otherwise return False.

BSearchRec(A, x):
1. if len(A) == 1:
2.    if A[0] == x:   return True
3.    else:   return False
4. mid = floor( len(A) / 2 )
5. if x < A[mid]:   return BSearchRec(A[0:mid], x)
6. else:   return BSearchRec(A[mid:len(A)], x)
```

PROOF: For $n \in \mathbb{N}$, define the predicate

$Q(n)$: If A is a non-empty sorted list of integers, x is an integer, and $n = \text{len}(A)$, then BSearchRec(A, x) returns True if x is in A and False otherwise.

We use the PCI to prove that $Q(n)$ holds for all $n > 0$. Correctness follows from this.

BASIS: Let $n = 1$. $A[0]$ is the only element in A. By lines 1–3, True is returned if $A[0] = x$ and False is returned otherwise, as desired.

INDUCTION STEP: Let $n > 1$. Suppose that $Q(j)$ holds whenever $1 \leq j < n$ (IH). We want to prove that $Q(n)$ holds.

By lines 1 and 4, $mid = \left\lfloor \dfrac{\text{len}(A)}{2} \right\rfloor$. We have $0 < mid < \text{len}(A)$ by PL2.2. Consider two cases: $x < A[mid]$ and $x \geq A[mid]$.

If $x < A[mid]$, $\text{len}\big(A[0\colon mid]\big) = mid$, which satisfies $0 < mid < \text{len}(A)$ by PL2.2. By the IH, BSearchRec$\big(A[0\colon mid]\,, x\big)$ on line 5 returns True if x is in A and False otherwise. Thus, $Q(n)$ holds in this case.

If $x \geq A[mid]$, $\text{len}\Big(A\big[mid\colon \text{len}(A)\big]\Big) = \text{len}(A) - mid$ satisfies $1 \leq \text{len}(A) - mid < n$ as $0 < mid < \text{len}(A) \implies 0 > -mid > -\text{len}(A)$ (by PL2.2). By the IH, BSearchRec$\Big(A\big[mid\colon \text{len}(A)\big]\,, x\Big)$ on line 6 returns True if x is in A and False otherwise, as desired. ■

Week 5 Regular Languages

5.1 Regular Languages

Definition 5.1 (Alphabet)
An alphabet Σ is a set of symbols (e.g. $\Sigma = \{0, 1\}$).

The set of all *finite* strings with symbols from Σ is Σ^*. We denote the empty string by $\epsilon \in \Sigma^*$.

Definition 5.2 (Language)
A *language* is a subset of Σ^* (e.g. $\emptyset$ and Σ^*).

Definition 5.3 (Language Operations)
For any languages L, L_1, and L_2, we have the following language operations:

- *Complement:* $\overline{L} = \Sigma^* - L = \{x \in \Sigma^* : x \notin L\}$
- *Union:* $L_1 \cup L_2$
- *Intersection:* $L_1 \cap L_2$
- *Concatenation:* $L_1 \cdot L_2 = \{xy : x \in L_1 \text{ and } y \in L_2\}$

  Note that $\emptyset \cdot L = \emptyset = L \cdot \emptyset$
- *Kleene Star:* $L^* = \{x : x = \epsilon \text{ or } x = y_1 \dots y_k \text{ where } k > 0 \text{ and each } y_i \in L\}$
- *Exponentiation:* $L^k = \begin{cases} \{\epsilon\} & \text{if } k = 0 \\ L^{k-1} \cdot L & \text{if } k > 0 \end{cases} \quad (k \in \mathbb{N})$

  Note that $L^* = \bigcup_{k=0}^{\infty} L^k$
- *Reversal:* $L^R = \{x^R : x \in L\}$

5.2 Regular Expressions

Definition 5.4 (Regular Expression)
Consider an alphabet Σ and strings in $\Sigma \cup \{+, \emptyset, \epsilon, *, (,)\}$. The *set of all regular expressions* $\mathcal{RE}$ (over Σ) is defined as the smallest set such that:

Basis: $\emptyset, \epsilon, a \in \mathcal{RE}$ for every $a \in \Sigma$.

Induction Step: If $R, S \in \mathcal{RE}$, then $(R + S), (RS), R^* \in \mathcal{RE}$.

Remark: *In definition 5.4, the + in $(R + S)$ refers to "or" (e.g. $(0 + 1)$ can be 0 or 1).*

Example 5.5
For the symbols $\emptyset, \epsilon, 0, 1$, the following strings are in $\mathcal{RE}$: $(0 + \emptyset)$, (01), ϵ^*, $(0 + \emptyset)^*$, and $(1(01))$. Note that $0 + 1 \notin \mathcal{RE}$.

Definition 5.6 (Semantics of Regular Expressions)
Let R be a regular expression and $s \in \Sigma^*$. R matches s or s matches R. The *language* of R, denoted by $\mathcal{L}(R)$, is the set of strings in Σ^* that match R. We define this set as follows.

Basis:

- $\mathcal{L}(\emptyset) = \emptyset$
- $\mathcal{L}(\epsilon) = \{\epsilon\}$
- $\mathcal{L}(a) = \{a\}$ for every $a \in \Sigma$.

Induction Step: Let $R, S \in \mathcal{RE}$.

- $\mathcal{L}\big((R + S)\big) = \mathcal{L}(R) \cup \mathcal{L}(S)$
- $\mathcal{L}\big((RS)\big) = \mathcal{L}(R) \cdot \mathcal{L}(S)$
- $\mathcal{L}(R^*) = \mathcal{L}(R)^*$

Remark: *In definition 5.6, the arguments for $\mathcal{L}$ are symbols, while the outputs are sets.*

Example 5.7
Find a regular expression R such that $\mathcal{L}(R) = \Sigma^*$ for $\Sigma = \{0, 1\}$.

Consider $R = (0 + 1)^*$. We have

$$\mathcal{L}(R) = \mathcal{L}\big((0+1)^*\big) = \mathcal{L}\big((0+1)\big)^* = \big(\mathcal{L}(0) \cup \mathcal{L}(1)\big)^* = \big(\{0\} \cup \{1\}\big)^* = \{0,1\}^* = \Sigma^*$$

Note that $R' = (0^*1^*)^*$ is also a solution.

Remark: *In general, there can be infinitely many solutions for problems involving finding a regular expression R such that $\mathcal{L}(R) = \Sigma^*$ given some alphabet Σ.*

Example 5.8
For $R = (00)^*$, find $\mathcal{L}(R)$.

We have $\mathcal{L}(R) = \big\{0^{2n} : n \in \mathbb{N}\big\}$.

Example 5.9
Find a regular expression R such that $\mathcal{L}(R) = \big\{0^{2n+1} : n \in \mathbb{N}\big\}$.

One solution is $R = (0(00)^*)$.

Example 5.10
Find R such that $\mathcal{L}(R) = \{1011\}$.

$R = ((10)(11))$ is a solution.

Definition 5.11 (Conventions for Language Operation Notation)
To avoid clutter, we omit parentheses (unless a particular order of operations is to be enforced) and use the following precedence of operations, from highest to lowest:

1. * (Kleene star)
2. $\cdot$ concatenation
3. $+$

When omitting parentheses with operators of equal precedence, we associate as follows:

- $+$ and $\cdot$: right associate (e.g. $\epsilon + 0 + 1$ is interpreted as $(\epsilon + (0 + 1))$).
- *: left associate

Definition 5.12 (Equivalence of Regular Expressions)
Two regular expressions R and S are *equivalent*, denoted by $R \equiv S$, if $\mathcal{L}(R) = \mathcal{L}(S)$.

Definition 5.13 (Regularity)
Let $L \subseteq \Sigma^*$ be a language. L is *regular* if there is a regular expression R such that $L = \mathcal{L}(R)$.

Example 5.14
$\{0^n 1^n : n \in \mathbb{N}\}$ is not regular.

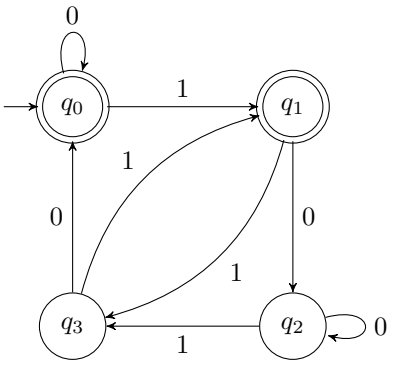
Definition 5.15 (Preservation of Regular Languages)
A language operation f *preserves regular languages* if for all regular languages L, $f(L)$ is regular.

The rest of the lecture was spent covering the "Structural Induction for Regular Expressions" handout.

Week 6 Finite State Automata

6.1 Deterministic Finite State Automata

Example 6.1
Let $\Sigma = \{0, 1\}$. Consider the following deterministic finite state automaton:



For the input 10011, we have the following transitions:

	1	0	0	1	1
q_0	q_1	q_2	q_2	q_3	q_1

Since q_1 is an accepting state, 10011 is accepted. For the input 0101, we have

	0	1	0	1
q_0	q_0	q_1	q_2	q_3

q_3 is not an accepting state, so 0101 is not accepted.

Definition 6.2 (Deterministic Finite State Automaton)
A *deterministic finite state automaton (DFSA)* M is a quintuple $M = (Q, \Sigma, \delta, s, F)$, where

- Q is a finite set of states
- Σ is the input alphabet
- $\delta \colon Q \times \Sigma \to Q$ is the transition function; $\delta(q, c) = q'$ denotes that when the symbol c is read in state q, the next state is q'
- $s \in Q$ is the initial state
- $F \subseteq Q$ is the set of accepting states

Definition 6.3 (DFSA Extended Transition Function)
Given a DFSA $M = (Q, \Sigma, \delta, s, F)$, the *extended transition function* is defined as

$$\delta^*(q, x) = \begin{cases} q & \text{if } x = \epsilon \\ \delta\big(\delta^*(q, y), a\big) & \text{if } x = ya \text{ for some } y \in \Sigma^* \text{ and } a \in \Sigma \end{cases}$$

Remark: *In definition 6.3, $\delta^* \colon Q \times \Sigma^* \to Q$ denotes that when the string x is read in state q, the next state is q'.*

Definition 6.4 (Language of a DFSA)
Given a DFSA $M = (Q, \Sigma, \delta, s, F)$, the *language of M* is

$$\mathcal{L}(M) = \big\{x \in \Sigma^* : \delta^*(s, x) \in F\big\}$$

Remark: *In definition 6.4, $\delta^*(s, x) \in F$ denotes that M accepts x.*

Example 6.5
Consider the DFSA in example 6.1. We will show that

$$\mathcal{L}(M) = \{x \in \Sigma^* : x \text{ has odd parity if and only if } x \text{ ends with } 1\}$$

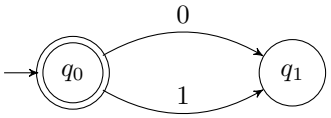
State Invariant (SI):

$$\delta^*(q_0, x) = \begin{cases} q_0 & \text{if and only if } x \text{ has even parity and } x \text{ does not end with 1} \\ q_1 & \text{if and only if } x \text{ has odd parity and } x \text{ ends with 1} \\ q_2 & \text{if and only if } x \text{ has odd parity and } x \text{ does not end with 1} \\ q_3 & \text{if and only if } x \text{ has even parity and } x \text{ ends with 1} \end{cases}$$

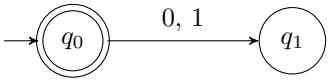
Prove this by induction on x, starting with $x = \epsilon$ for the basis, then letting $x = ya$ for some $a \in \Sigma$ and $y \in \Sigma^*$ (for which the SI holds) in the induction step.

Remark: *In this course, we are not expected to prove state invariants due to their repetitiveness over several cases.*

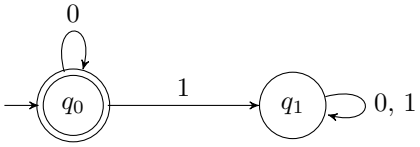
Definition 6.6 (FSA Diagram Conventions)
When there are multiple transitions from one state to another, instead of drawing separate arrows for each, a single arrow is drawn with comma-separated symbols as the label. For example, instead of drawing



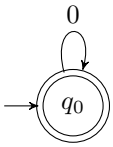
we would draw



Furthermore, dead states are omitted. For example, the diagram



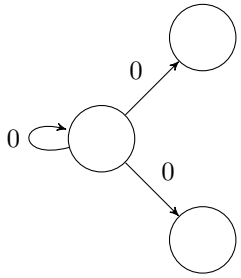
would instead be drawn as



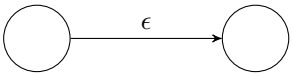
6.2 Nondeterministic Finite State Automata

A nondeterministic finite state automaton (NFSA) is like a DFSA with two added features:

1. Multiple possible transitions when reading the same symbol

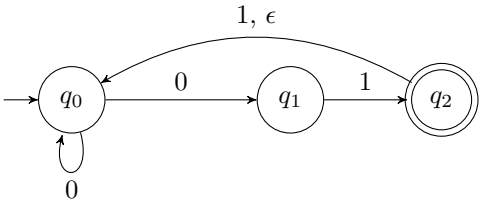


2. ϵ-transitions (without reading any input symbol)



An NFSA *accepts* a string x if by starting at the initial state, there is a way to read all of x and end in an accepting state.

Example 6.7
Consider the following NFSA:



For the input 01, we may have any of

	0	1
q_0	q_0	q_{dead}

	0	1	ϵ
q_0	q_1	q_2	q_0

	0	1
q_0	q_1	q_2

Note that the first and second paths do not result in an accepting state, while the third one does. Thus, 01 is accepting.

Definition 6.8 (Nondeterministic Finite State Automaton)
A *nondeterministic finite state automaton (NFSA)* is a quintuple $M = (Q, \Sigma, \delta, s, F)$, where

- Q is a finite set of states
- Σ is the input alphabet
- $\delta\colon Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q)$ is the transition function; $\delta(q, c) = q'$ denotes that when the symbol c is read in state q, the next state is q'
- $s \in Q$ is the initial state
- $F \subseteq Q$ is the set of accepting states

Remark: *$\mathcal{P}(Q)$ denotes the* power set *of Q.*

Definition 6.9 (NFSA Extended Transition Function)
Let $M = Q, \Sigma, \delta, s, F$ be an NFSA and $\mathcal{E}(q)$ denote the set of all states that are reachable from q using only ϵ-transitions.

The *extended transition function* $\delta^*\colon Q \times \Sigma^* \to \mathcal{P}(Q)$ of M is defined as

$$\delta^*(q, x) = \begin{cases} \mathcal{E}(q) & \text{if } x = \epsilon \\ \displaystyle\bigcup_{q' \in \delta^*(q,y)} \left(\bigcup_{q'' \in \delta(q', a)} \mathcal{E}(q'') \right) & \text{if } x = ya \text{ for some } y \in \Sigma^* \text{ and } a \in \Sigma \end{cases}$$

Remark: *Informally, in definition 6.9, $\delta^*(q, x)$ is the set of states that are reachable from state q reading all of the string x.*

Definition 6.10 (Language of a NFSA)
The *language* of a NFSA M is defined as

$$\mathcal{L}(M) = \left\{ x \in \Sigma^* : \delta^*(s, x) \cap F \neq \emptyset \right\}$$

Remark: *In definition 6.10, $\delta^*(s, x) \cap F \neq \emptyset$ denotes that M accepts x.*

6.3 THE BIG RESULT

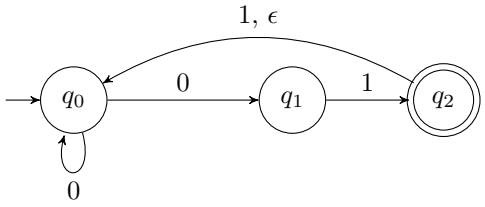
Theorem 6.11 (THE BIG RESULT)
Let L be a language. The following statements are equivalent:

1. *$L = \mathcal{L}(R)$ for some regular expression R (i.e. L is regular)*
2. *$L = \mathcal{L}(M)$ for some DFSA M*
3. *$L = \mathcal{L}(M)$ for some NFSA M*

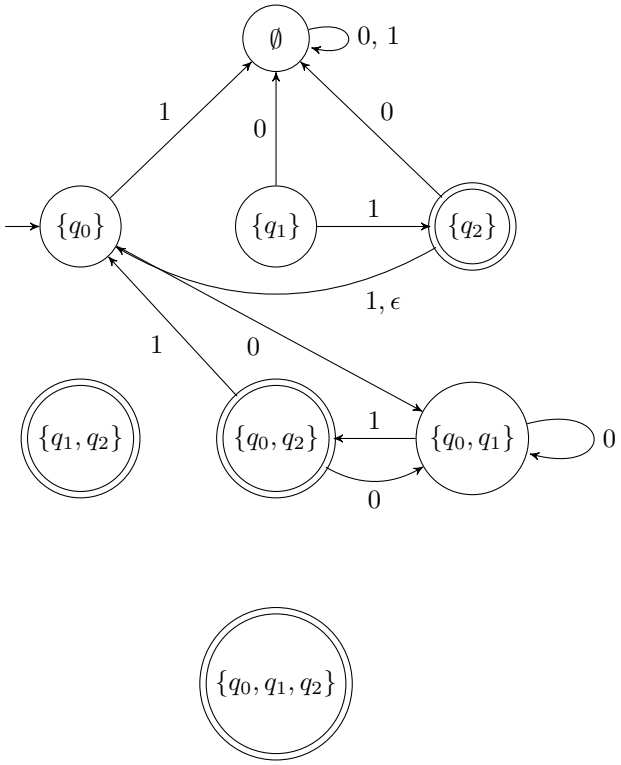
Week 7 Finite State Automata Continued

7.1 Finite State Automata

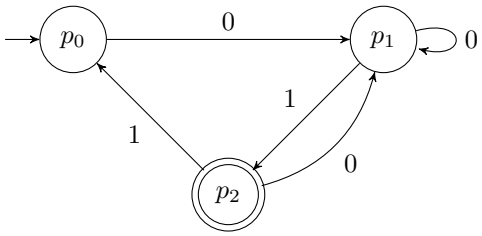
Example 7.1 (Subset Construction)
Consider the following NFSA M (as in example 6.7):



Given the NFSA M, construct a DFSA M' such that $\mathcal{L}(M') = \mathcal{L}(M)$.



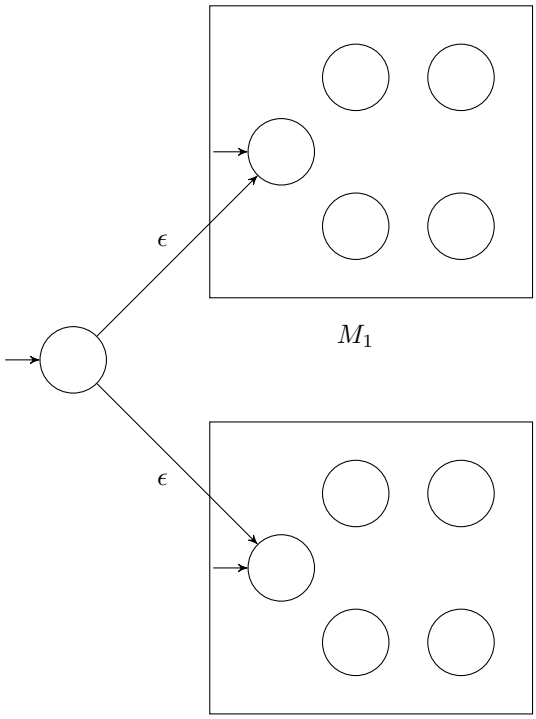
A simplified version of this is as follows:



Theorem 7.2 (Closure Properties with FSAs)
The complementation, union, intersection, concatenation, and Kleene star of any regular lanaguages (or equivalently, FSAs) is also regular.

Complementation: *Suppose that $L = \mathcal{L}(M)$, where $M = (Q, \Sigma, \delta, s, F)$ is a DFSA. The DFSA $\overline{M} = (Q, \Sigma, \delta, s, Q - F)$ satisfies $\mathcal{L}(\overline{M}) = \overline{\mathcal{L}(M)}$ (note that $\overline{F} = Q - F$).*

Union: *Let M_1 and M_2 be DFSAs. The NFSA $M_\cup$ whose diagram is below satisfies $\mathcal{L}(M_\cup) = \mathcal{L}(M_1) \cup \mathcal{L}(M_2)$:*

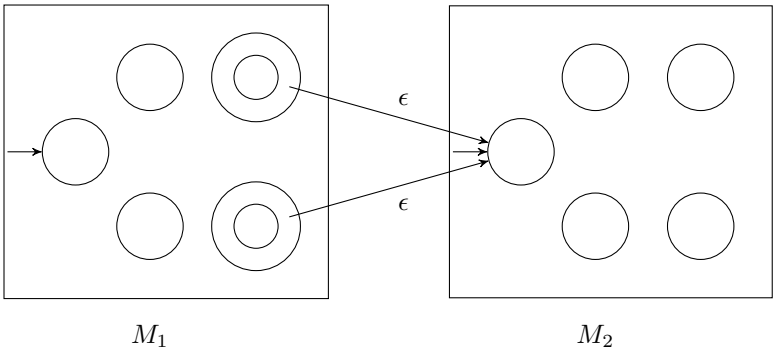


Intersection: *Suppose that* $M_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$ *and* $M_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$ *are DFSAs. Consider the DFSA* $M_\cap = (Q_\cap, \Sigma, \delta_\cap, s_\cap, F_\cap)$*, where*

- $Q_\cap = Q_1 \times Q_2$
- $\delta_\cap\big((q_1, q_2), c\big) = \big(\delta_1(q_1, c), \delta_2(q_2, c)\big)$ *for all* $(q_1, q_2) \in Q_\cap$ *and* $c \in \Sigma$
- $s_\cap = (s_1, s_2)$
- $F_\cap = F_1 \times F_2$

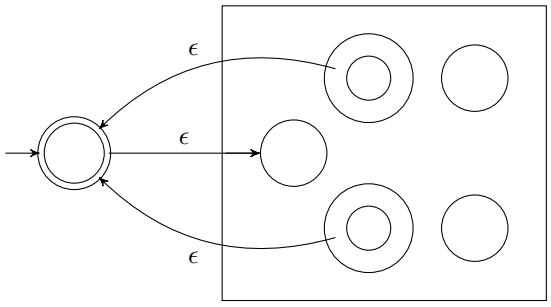
$M_\cap$ *satisfies* $\mathcal{L}(M_\cap) = \mathcal{L}(M_1) \cap \mathcal{L}(M_2)$.

Concatenation: *Let* M_1 *and* M_2 *be DFSAs. The NFSA* $M.$ *whose diagram is given below satisfies* $\mathcal{L}(M.) = \mathcal{L}(M_1) \cdot \mathcal{L}(M_2)$*:*



The accepting states in M_1 *are only accepting for* M_1*, not for* $M.$.

Kleene Star: *Let* M *be a DFSA. The NFSA* M^* *whose diagram is given below satisfies* $\mathcal{L}(M^*) = \mathcal{L}(M)^*$*:*



M

The accepting states in M are only accepting for M, not M^*.

Remark: *In theorem 7.2, the intersection property can be proven without constructing a FSA. Suppose that L_1 and L_2 are regular. Since $L_1 \cap L_2 = \overline{\left(\overline{L_1} \cup \overline{L_2}\right)}$, $L_1 \cap L_2$ is regular using the complementation and union closure properties.*

Also, the FSA construction used for the intersection property is known as the Cartesian product construction.

Lemma 7.3 (The Pumping Lemma)
Let L be a regular language. There exists some $n \in \mathbb{Z}^+$ such that for any $x \in L$ with $|x| \geq n$, there are strings u, v, and w such that each of the following conditions hold:

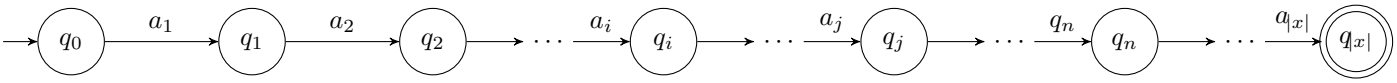
(i) $x = uvw$

(ii) $v \neq \epsilon$

(iii) $|uv| \leq n$

(iv) $uv^k w \in L$ *for all* $k \in \mathbb{N}$

Proof: Suppose that L is a regular language. There exists a DFSA M such that $L = \mathcal{L}(M)$. Now let n be the number of states in M. Suppose that $x \in L$ with $|x| \geq n$. Consider the states in M as x is read by M.



By the pigeonhole principle, there exist $i, j \in \mathbb{N}$ such that $0 \leq i < j \leq n$ such that $q_i = q_j$. Let $u = a_1 \dots a_i$, $v = a_{i+1} \dots a_j$, and $w = a_{j+1} \dots a_{|x|}$.

We have $x = a_1 \dots a_{|x|} = a_1 \dots a_i a_{i+1} \dots a_j a_{j+1} \dots a_{|x|} = uvw$, so (i) holds. Since $i < j$, $i+1 \leq j$, so $v = a_{i+1} \dots a_j \neq \epsilon$. This proves (ii). Furthermore, $|uv| = \left|a_1 \dots a_i a_{i+1} \dots a_j\right| = j \leq n$, so (iii) holds. We will now show that (iv) holds.

Now $\delta^*(q_i, v) = \delta^*(q_i, a_{i+1} \dots a_j) = q_j = q_i$, using the fact that $q_i = q_j$. Thus, $\delta^*(q_i, v^k) = q_i = q_j$ for all $k \in \mathbb{N}$. We have $\delta^*(q_j, w) = q_{|x|}$ and $\delta^*(q_0, u) = q_i$, so $\delta^*(q_0, uv^k w) = q_{|x|}$. Since $\delta^*(q_0, x) = q_{|x|}$ and $x \in \mathcal{L}(M)$, $q_{|x|}$ is an accepting state. Therefore, $uv^k w \in \mathcal{L}(M) = L$ for all $k \in \mathbb{N}$, proving (iv). ∎

The rest of the lecture was spent covering the "proving closure property using FSA method" and "proving languages not regular using Pumping Lemma" handouts.

Week 8 Context-Free Grammars

8.1 Context-Free Grammars

Example 8.1
Consider the following context-free grammar:

$$\begin{aligned} S &\to ABC \\ A &\to 11A \\ A &\to \epsilon \\ B &\to 0B0 \\ B &\to 0 \\ C &\to C10 \\ C &\to C01 \\ C &\to 1 \end{aligned}$$

Here is a string that can be generated using this grammar:

$$S \Rightarrow ABC \Rightarrow 11ABC \Rightarrow 11BC \Rightarrow 110B0C \Rightarrow 11000C \Rightarrow 11000C10 \Rightarrow 11000C0110 \Rightarrow 1100010110$$

Remark: *With context-free grammars, a convention is to write each production of a variable on one line rather than having a separate line for each one. Using this convention, example 8.1 would become*

$$\begin{aligned} S &\to ABC \\ A &\to 11A, \epsilon \\ B &\to 0B0, 0 \\ C &\to C10, C01, 1 \end{aligned}$$

Definition 8.2 (Context-Free Grammar)
A *context-free grammar (CFG)* is a 4-tuple $G = (V, \Sigma, P, S)$, where

- V is a finite set of variables
- Σ is a finite set of terminals
- P is a finite set of productions; each production is of the form

  $$A \to \alpha$$

  where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$
- $S \in V$ is the start variable

Remark: *In definition 8.2, the set of terminals Σ is similar to an alphabet in FSAs.*

Definition 8.3 (Derivation)
For $\alpha, \beta \in (V \cup \Sigma)^*$, $\alpha \Rightarrow \beta$ means that there exist $\alpha_1, \alpha_2, \gamma \in (V \cup \Sigma)^*$ and $A \in V$ such that

$$\alpha = \alpha_1 A \alpha_2 \qquad \beta = \alpha_1 \gamma \alpha_2 \qquad A \to \gamma \in P$$

β can be *derived* from α, denoted $\alpha \Rightarrow^* \beta$, if there exists a sequence $\alpha_1, \ldots, \alpha_k \in (V \cup \Sigma)^*$ such that

$$\alpha = \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_k = \beta$$

Remark: *For all $\alpha \in (V \cup \Sigma)^*$, $\alpha \Rightarrow^* \alpha$.*

Definition 8.4 (Language of a Context-Free Grammar)
The *language* of a CFG $G = (V, \Sigma, P, S)$ is

$$\mathcal{L}(G) = \{x \in \Sigma^* : S \Rightarrow^* x\}$$

Example 8.5
Consider the CFG from example 8.1, given by

$$S \to ABC$$
$$A \to 11A, \epsilon$$
$$B \to 0B0, 0$$
$$C \to C10, C01, 1$$

A generates $\mathcal{L}\big((11)^*\big)$, B generates $\mathcal{L}\big(0(00^*)\big)$, and C generates $\mathcal{L}\big(1(10+01)^*\big)$. Thus,

$$\mathcal{L}(G) = \mathcal{L}\big((11)^*0(00)^*1(10+01)^*\big)$$

Definition 8.6 (Context-Free Language)
A *context-free language (CFL)* is a language L such that

$$\mathcal{L}(G) = L$$

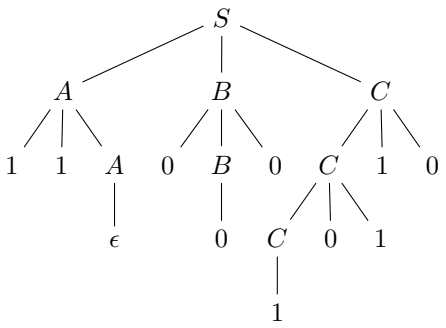
for some CFG G.

Example 8.7
Consider the CFG given by $S \to 1S0, \epsilon$. S generates $\{1^n0^n : n \in \mathbb{N}\}$.

Example 8.8 (Parse Tree)
Consider the CFG in example 8.1. Recall that

$$S \Rightarrow ABC \Rightarrow 11ABC \Rightarrow 11BC \Rightarrow 110B0C \Rightarrow 11000C \Rightarrow 11000C10 \Rightarrow 11000C0110 \Rightarrow 1100010110$$

This can be represented by the following parse tree:



Definition 8.9 (Context-Free Grammar Ambiguity)
A CFG G is *ambiguous* if there exists some $x \in \mathcal{L}(G)$ with two different parse trees for its derivation.

Remark: *It was noted during lecture that definition 8.9 is not rigorous.*

Definition 8.10 (Context-Free Language Inherent Ambiguity)
A CFL is *inherently ambiguous* if every CFG G that generates it is ambiguous.

Remark: "Ambiguity" *refers only to context-free grammars, while* "inherent ambiguity" *refers only to context-free languages.*

Definition 8.11
Given $x, y \in \Sigma^*$, define

$$\#_y(x) = \Big|\big\{(u, v) : x = uyv\big\}\Big|$$

Remark: *In definition 8.11,* $\#_y(x)$ *is the* number of occurrences *of* y *in* x.

Example 8.12
$\#_0(x)$ is the number of 0s in x, and $\#_1(x)$ is the number of 1s in x.

Example 8.13 (CFG Design — Example 8.3)
Let $L = \big\{x \in \{0,1\}^* : \#_0(x) = \#_1(x)\big\}$. Find a CFG G such that $\mathcal{L}(G) = L$.

The design is as follows:

- S generates L
- A generates $\left\{x \in \{0,1\}^* : \#_0(x) = \#_1(x) + 1\right\}$
- B generates $\left\{x \in \{0,1\}^* : \#_1(x) = \#_0(x) + 1\right\}$

This yields

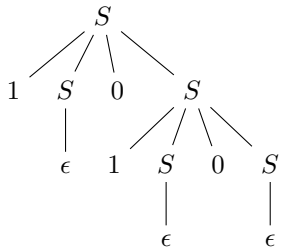
$$S \to \epsilon, 1A, 0B$$
$$A \to 0S, 1AA$$
$$B \to 1S, 0BB$$

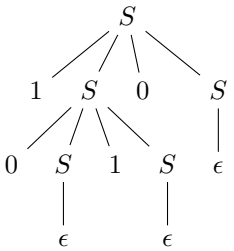
Note that the above design is unambiguous. An alternative, simpler design is: S generates L with the productions

$$S \to \epsilon, 1S0S, 0S1S$$

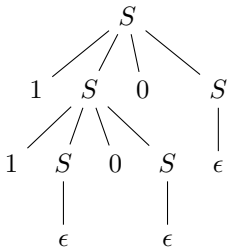
This design is ambiguous, unlike the first one.

For $x = 1010$, here are two possible parse trees:





For $x = 1100$, one possible parse tree is



8.2 Context-Free and Regular Languages

Theorem 8.14 (Regular Languages are Context-Free)
Every regular language is context-free.

PROOF: A sketch of a proof by structural induction is as follows.

Define a predicate P on $\mathcal{RE}$ by

$P(R)$: there exists a CFG $G = (V, \Sigma, P, S)$ such that $\mathcal{L}(G) = \mathcal{L}(R)$

BASIS: There are three base cases:

1. $R = \emptyset$: Let G be such that $P = \emptyset$
2. $R = \epsilon$: Let G be $S \to \epsilon$
3. $R = a$, where $a \in \Sigma$: Let G be $S \to a$

Induction Step: Let R_1 and R_2 be regular expressions. Suppose that we have $G_1 = (V_1, \Sigma, P_1, S_1)$ and $G_2 = (V_2, \Sigma, P_2, S_2)$ such that $\mathcal{L}(R_1) = \mathcal{L}(G_1)$ and $\mathcal{L}(R_2) = \mathcal{L}(G_2)$ (IH). We consider three cases.

1. $R = R_1 + R_2$: Let $V = V_1 \cup V_2 \cup \{S\}$ for some new start variable S and $P = P_1 \cup P_2 \cup \{S \to S_1, S \to S_2\}$
2. $R = R_1R_2$: Let $V = V_1 \cup V_2 \cup \{S\}$ for some new start variable S and $P = P_1 \cup P_2 \cup \{S \to S_1S_2\}$
3. $R = R_1^*$: Let $V = V_1 \cup \{S\}$ for some new start variable S and $P = P_1 \cup \{S \to \epsilon, S \to SS_1\}$

■

Definition 8.15 (Grammar Right-Linearity)
A CFG $G = (V, \Sigma, P, S)$ is *right linear* if every production in P has one of the following forms:

1. $A \to \epsilon$
2. $A \to xB$

where $A, B \in V$ and $x \in \Sigma^*$.

G is *strict right-linear* if every production in P has one of following forms:

1. $A \to \epsilon$
2. $A \to xB$

where $A, B \in V$, $x \in \Sigma^*$, and $|x| \leq 1$.

Theorem 8.16 (Strict Right-Linearity and Regularity Equivalence)
If L is a language, then L is regular if and only if $L = \mathcal{L}(G)$ for some strict right-linear CFG G.

PROOF: A sketch of a proof is as follows.

Let $M = (Q, \Sigma, \delta, s, F)$ be an FSA with $L = \mathcal{L}(M)$. Consider the strict right-linear CFG $G = (V, \Sigma, P, S)$, where $Q = V$, $s = S$, and $F = \{A : A \to \epsilon \in P\}$. G satisfies $L = \mathcal{L}(G)$. ■

Theorem 8.17
Let G be a right-linear CFG. There exists a strict right-linear CFG G' such that $\mathcal{L}(G') = \mathcal{L}(G)$.

PROOF: The main idea for this proof is to replace each production of the form $A \to a_1 a_2 \dots a_k B$ with

$$\begin{aligned} A &\to a_1 B_1 \\ A &\to a_2 B_2 \\ &\vdots \\ A &\to a_k B_k \end{aligned}$$

■

8.3 THE BIG RESULT Plus

Theorem 8.18 (THE BIG RESULT Plus)
Let L be a language. The following statements are equivalent:

1. *$L = \mathcal{L}(R)$ for some regular expression R*
2. *$L = \mathcal{L}(M)$ for some DFSA M*
3. *$L = \mathcal{L}(M)$ for some NFSA M*
4. *$L = \mathcal{L}(G)$ for some right-linear CFG G*
5. *$L = \mathcal{L}(G)$ for some strict right-linear CFG G*

Week 9 Pushdown Automata

9.1 Context-Free Languages Continued

Example 9.1
Consider the language $L = \{x \in \Sigma^* : \#_{00}(x) = \#_{11}(x)\}$ over $\Sigma = \{0, 1\}$. Find a CFG G such that $\mathcal{L}(G) = L$.

Consider the following design:

- S generates L
- A_{00} generates $\{x \in \Sigma^* : x \in L, x \text{ starts with } 0, \text{and } x \text{ ends with } 0\}$
- A_{01} generates $\{x \in \Sigma^* : x \in L, x \text{ starts with } 0, \text{and } x \text{ ends with } 1\}$
- A_{10} generates $\{x \in \Sigma^* : x \in L, x \text{ starts with } 1, \text{and } x \text{ ends with } 0\}$
- A_{11} generates $\{x \in \Sigma^* : x \in L, x \text{ starts with } 1, \text{and } x \text{ ends with } 1\}$

The CFG is as follows:

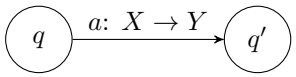
$$\begin{aligned} S &\to \epsilon, A_{00}, A_{01}, A_{10}, A_{11} \\ A_{00} &\to 0A_{10}, 0A_{01}A_{10}, 0 \\ A_{01} &\to 0A_{11}, 0A_{01}A_{11} \\ A_{10} &\to 1A_{00}, 1A_{10}A_{00} \\ A_{11} &\to 1A_{01}, 1A_{10}A_{01}, 1 \end{aligned}$$

Remark: *In example 9.1, the productions were created by considering separate cases for the symbol that the string begins and ends with. For each of these cases (A_{00}, A_{01}, A_{10}, and A_{11}), the second symbol in the string (if any) was considered case-by-case; the number of* 00*s and* 11*s were then "balanced" by adding additional* 00*s and/or* 11*s as needed.*

9.2 Pushdown Automata

Definition 9.2 (Pushdown Automaton)
A *pushdown automaton* (PDA) is a 6-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, where

- Q is a set of states
- Σ is the input alphabet
- Γ is the stack alphabet
- $\delta\colon Q \times \big(\Sigma \cup \{\epsilon\}\big) \times \big(\Gamma \cup \{\epsilon\}\big) \to \mathcal{P}\Big(Q \times \big(\Gamma \cup \{\epsilon\}\big)\Big)$ is the transition function; each label is of the form



  where $a \in \Sigma \cup \{\epsilon\}$, $X, Y \in \Gamma \cup \{\epsilon\}$, and $q, q' \in Q$. There are four cases with X and Y:

  1. $X = \epsilon$ and $Y \in \Gamma$: push Y onto the stack
  2. $X \in \Gamma$ and $Y = \epsilon$: pop X from the stack
  3. $X, Y \in \Gamma$: replace X with Y at the top of the stack
  4. $X = Y = \epsilon$: perform no action on the stack
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the set of accepting states

Remark: *Informally, a PDA is a NFSA with a stack.*

Definition 9.3 (PDA Acceptance)
A PDA M *accepts* the input $x \in \Sigma^*$ if by starting with an empty stack, there is a sequence of transitions that:

- Reads all of x

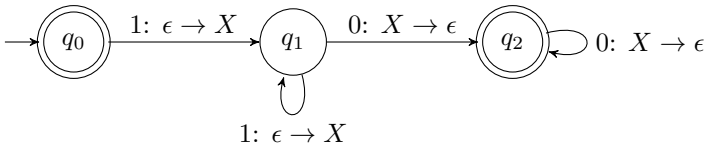
- Ends in an accepting state and
- Leaves the stack empty

Definition 9.4 (Language of a PDA)
The *language* of a PDA M is

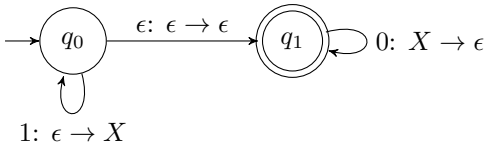
$$\mathcal{L}(M) = \{x \in \Sigma^* : M \text{ accepts } x\}$$

Example 9.5
Let $\Sigma = \{0, 1\}$ and $L = \{1^n 0^n : n \in \mathbb{N}\}$. Find a PDA that accepts L.

A PDA that accepts L is as follows:



Note that the above PDA is deterministic. Another possible PDA is

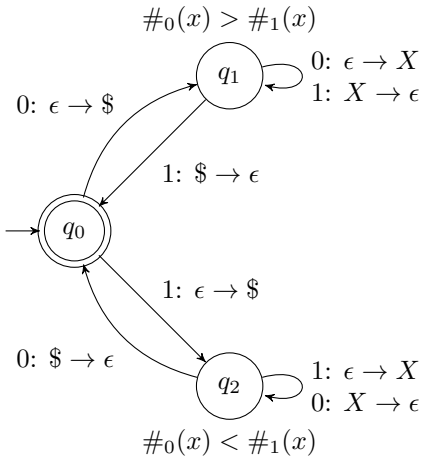


Definition 9.6 (Deterministic and Non-Deterministic PDA)
A *deterministic pushdown automaton* (DPDA) is a PDA where each input string has only one possible sequence of transitions.

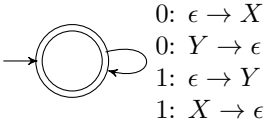
A *non-deterministic pushdown automaton* is a PDA where some input string may have more than one possible sequence of transitions.

Remark: *Definition 9.6 was not explicitly stated during lecture; it was informally explained. As a result, the definition is not rigorous.*

Example 9.7
Consider $L = \{x \in \Sigma^* : \#_0(x) = \#_1(x)\}$. A DPDA that accepts L is as follows:



A nondeterministic PDA that accepts L is



We will show that the above nondeterministic PDA M satisfies $\mathcal{L}(M) = L$.

If M accepts x, the stack must be empty after all transitions when reading x are complete. Thus, the transition $0\colon \epsilon \to X$ is performed the same number of times as $1\colon X \to \epsilon$, and the transition $1\colon \epsilon \to Y$ is performed the same number of times as $0\colon Y \to \epsilon$. It follows that $\#_0(x) = \#_1(x)$, so $x \in L$. Therefore, $\mathcal{L}(M) \subseteq L$. It can be shown using a tedious induction proof that $L \subseteq \mathcal{L}(M)$.

9.3 THE BIG RESULT II (Sequel)

Theorem 9.8 (THE BIG RESULT II)
Let L be a language. $L = \mathcal{L}(G)$ for some CFG G if and only if $L = \mathcal{L}(M)$ for some PDA M.

Example 9.9 (Non-Context-Free Language)
Consider $\{1^n 01^n : n \in \mathbb{N}\}$. This is not context-free.

Remark: *The proof for example 9.9 requires a pumping lemma for CFLs, where instead of "pumping" v in $x = uvw$, we pump v and y in $x = uvwyz$.*

Example 9.10
Find a CFG for $L_d = \left\{x \in \{0,1\}^* : 2\#_{00}(x) = \#_{11}(x)\right\}$.

Consider the following design:

- S generates L_d
- A_{00} generates $\{x \in L_d : x$ starts with 0 and ends with $0\}$
- A_{01} generates $\{x \in L_d : x$ starts with 0 and ends with $1\}$
- A_{10} generates $\{x \in L_d : x$ starts with 1 and ends with $0\}$
- A_{11} generates $\{x \in L_d : x$ starts with 1 and ends with $1\}$

The CFG is as follows:

$$\begin{aligned} S &\to \epsilon, A_{00}, A_{01}, A_{10}, A_{11} \\ A_{00} &\to 0, 0A_{10}, 0A_{01}A_{11}A_{10} \\ A_{01} &\to 0A_{11}, 0A_{01}A_{11}A_{11} \\ A_{10} &\to 1A_{00}, 1A_{11}A_{10}A_{00}, 1A_{10}A_{01}A_{10} \\ A_{11} &\to 1, 1A_{01}, 1A_{11}A_{10}A_{01}, 1A_{10}A_{01}A_{11} \end{aligned}$$

Example 9.11
Find a CFG $L_a = \left\{x \in \{0,1\}^* : \#_{00}(x) = \#_{11}(x) + 1\right\}$.

Consider a similar design as in example 9.10:

- S generates L_a
- A_{00} generates $\{x \in L_a : x$ starts with 0 and ends with $0\}$
- A_{01} generates $\{x \in L_a : x$ starts with 0 and ends with $1\}$
- A_{10} generates $\{x \in L_a : x$ starts with 1 and ends with $0\}$
- A_{11} generates $\{x \in L_a : x$ starts with 1 and ends with $1\}$

If we were to use this design, an attempt at creating a CFG would be

$$\begin{aligned} S &\to \epsilon, A_{00}, A_{01}, A_{10}, A_{11} \\ A_{00} &\to 0A_{10}, 0A_{01}A_{10} \end{aligned}$$

This design does not produce a CFG as intended, since $A_{00} \to 0A_{01}A_{10}$ has only one additional 11 but three additional 00s (this issue persists in general). With this in mind, we consider the following revised design using the language $L_e = \left\{x \in \Sigma^* : \#_{00}(x) = \#_{11}(x)\right\}$ from example 9.1:

- S generates L_a
- A_{00} generates $\{x \in L_e : x$ starts with 0 and ends with $0\}$
- A_{01} generates $\{x \in L_e : x$ starts with 0 and ends with $1\}$
- A_{10} generates $\{x \in L_e : x$ starts with 1 and ends with $0\}$
- A_{11} generates $\{x \in L_e : x$ starts with 1 and ends with $1\}$

We thus get the following CFG:

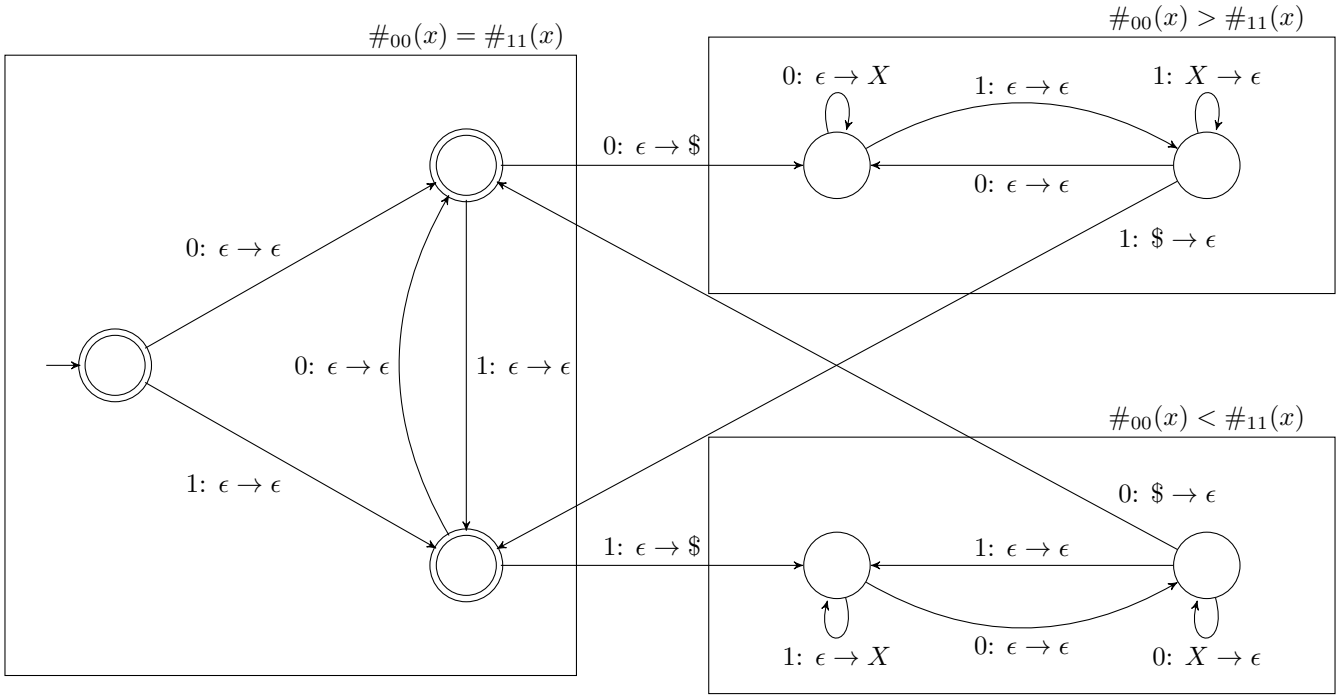
$$\begin{aligned} S &\to A_{00}A_{00}, A_{10}A_{00}, A_{00}A_{01}, A_{10}A_{01} \\ A_{00} &\to 0A_{10}, 0A_{01}A_{10}, 0 \\ A_{01} &\to 0A_{11}, 0A_{01}A_{11} \\ A_{10} &\to 1A_{00}, 1A_{10}A_{00} \\ A_{11} &\to 1A_{01}, 1A_{10}A_{01}, 1 \end{aligned}$$

This uses the same productions for A_{00}, A_{01}, A_{10}, and A_{11} as in example 9.1.

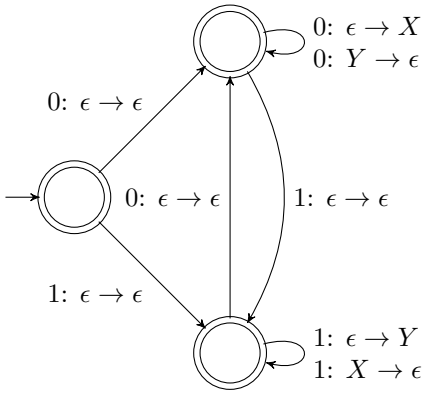
Week 10 Pushdown Automata Continued

10.1 Pushdown Automata

Example 10.1
Consider $L_e = \{x : \#_{00}(x) = \#_{11}(x)\}$ over $\Sigma = \{0, 1\}$. Using $L_0 = \{x \in \Sigma^* : \#_0(x) = \#_1(x)\}$ (as in example 9.7), we will construct PDAs accepting L_e. A DPDA is as follows:



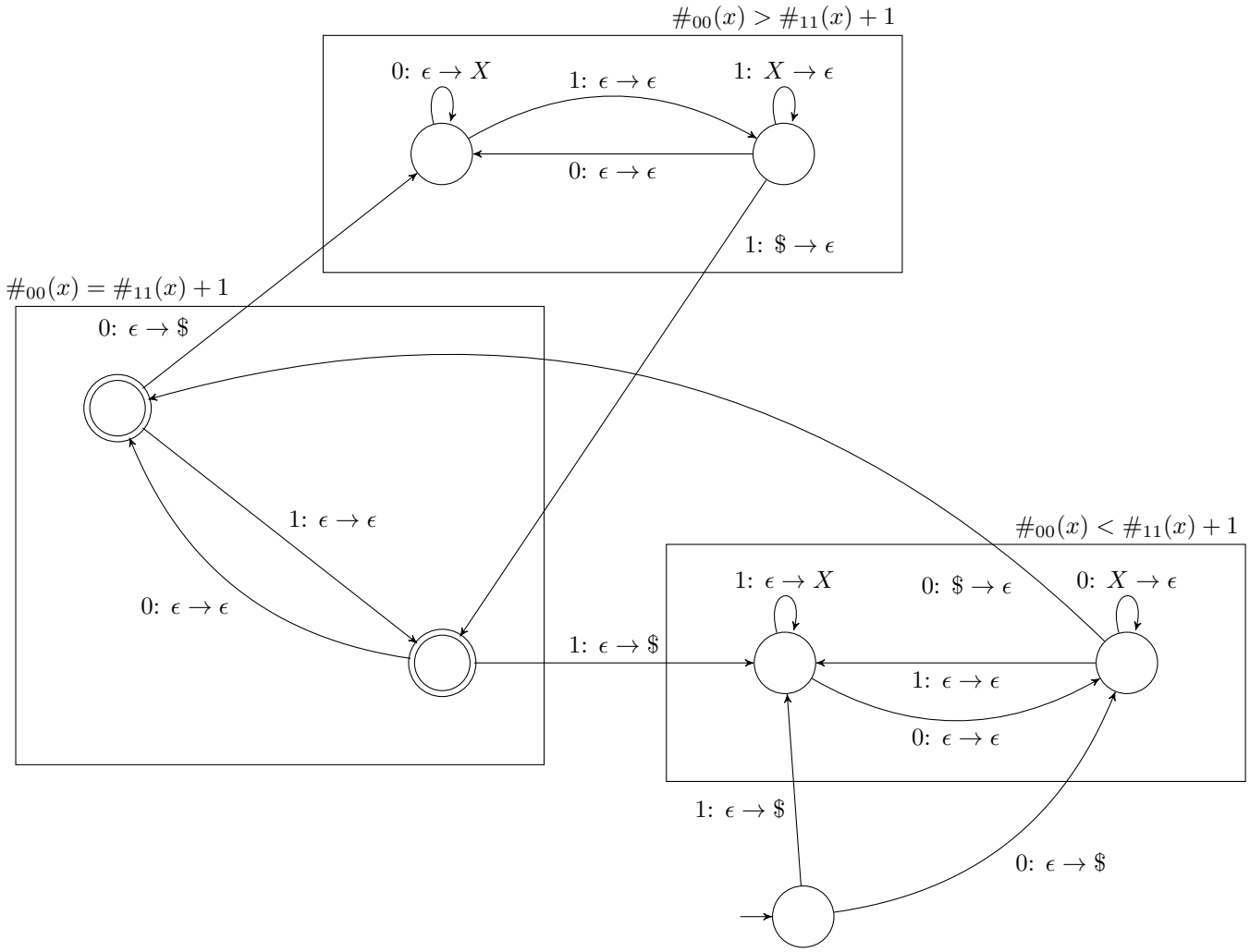
A NPDA is as follows:



Example 10.2
Let $L_d = \{x : 2\#_{00}(x) = \#_{11}(x)\}$. We will construct PDAs accepting L_d.

Insert PDAs here.

Example 10.3
Consider $L_a = \{x : \#_{00}(x) = \#_{11}(x) + 1\}$. We will construct PDAs for L_a using $L_e = \{x : \#_{00}(x) = \#_{11}(x)\}$ from example 10.1. A DPDA using *pre-processing* on the DPDA from example 10.1 is as follows:



Insert NPDA here.

Week 11 Propositional Logic

11.1 The Language of Propositional Logic

Definition 11.1 (Propositional Variables)
The set of *propositional variables* is denoted by PV.

Definition 11.2 (Propositional Formulas)
Let PV be a set of variables. The *set of propositional formulas* $\mathcal{F}_{PV}$ is the smallest set such that:

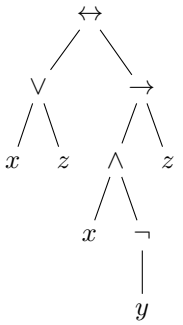
Basis: If $x \in PV$, then $x \in \mathcal{F}_{PV}$.

Induction Step: If $P, Q \in \mathcal{F}_{PV}$, then $\neg P, (P \wedge Q), (P \vee Q), (P \to Q), (P \leftrightarrow Q) \in \mathcal{F}_{PV}$.

Example 11.3 (Propositional Formula Parse Tree)
Consider the propositional formula

$$\Big((x \vee z) \leftrightarrow \big((x \wedge \neg y) \to z\big)\Big) \in \mathcal{F}_{PV}$$

where $x, y, z \in PV$. The *parse tree* for this formula is as follows:



The internal nodes are connectives, and the leaf nodes are elements of PV.

Definition 11.4 (Conventions for Propositional Formulas)
The following conventions are used when denoting propositional formulas:

1. The outermost pair of parentheses can be omitted; $P \to Q$ is shorthand for $(P \to Q)$
2. Binary connectives of the same precedence are *right associative*; $(P \to Q \to R)$ is shorthand for $\big(P \to (Q \to R)\big)$
3. The precedence of the connectives from highest to lowest is as follows:
   - $\neg$
   - $\wedge$
   - $\vee$
   - $\to$ and $\leftrightarrow$ at the same precedence

Definition 11.5 (Truth Assignment)
A *truth assignment* to PV is a function $\tau\colon PV \to \{0, 1\}$.

Definition 11.6 (Extended Truth Assignment)
For a truth assignment $\tau\colon PV \to \{0, 1\}$, the *extended truth assignment* is the function $\tau^*\colon \mathcal{F}_{PV} \to \{0, 1\}$ defined as follows:

Basis: If $x \in PV$, then $\tau^*(x) = \tau(x)$

Induction Step: If $P, Q \in \mathcal{F}_{PV}$, then

$$\tau^*(\neg P) = \begin{cases} 1 & \text{if } \tau^*(P) = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\tau^*(P \wedge Q) = \begin{cases} 1 & \text{if } \tau^*(P) = \tau^*(Q) = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\tau^*(P \vee Q) = \begin{cases} 1 & \text{if } \tau^*(P) = 1 \text{ or } \tau^*(Q) = 1 \\ 0 & \text{otherwise} \end{cases}$$

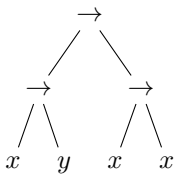
$$\tau^*(P \to Q) = \begin{cases} 1 & \text{if } \tau^*(P) = 0 \text{ or } \tau^*(Q) = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\tau^*(P \leftrightarrow Q) = \begin{cases} 1 & \text{if } \tau^*(P) = \tau^*(Q) \\ 0 & \text{otherwise} \end{cases}$$

Example 11.7 (Truth Table)
A *truth table* for the propositional formula $\big((x \to y) \to (x \to x)\big)$ is as follows:

x	y	$(x \to y)$	$\to$	$(x \to x)$
0	0	1	1	1
0	1	1	1	1
1	0	0	1	1
1	1	1	1	1

The parse tree for the formula is



Example 11.8
A truth table for $x \wedge \neg(y \to x)$ is as follows:

x	y	x	$\wedge$	$\neg$	$(y \to x)$
0	0	0	0	1	
0	1	0	1	0	
1	0	0	0	1	
1	1	0	0	1	

11.2 Satisfiability

Definition 11.9 (Satisfiability)
For $\tau\colon PV \to \{0,1\}$ and $P \in \mathcal{F}_{PV}$,

- τ *satisfies* P if and only if $\tau^*(P) = 1$
- τ *falsifies* P if and only if $\tau^*(P) = 0$

Similarly,

- P is a *tautology* if and only if for any truth assignment τ, τ satisfies P
- P is *satisfiable* if and only if for some truth assignment τ, τ satisfies P
- P is *falsifiable* if and only if for some truth assignment τ, τ falsifies P
- P is a *contradiction* if and only if for any truth assignment τ, τ falsifies P

Example 11.10
For $P = (x \to y)$, if $\tau(x) = \tau(y) = 1$, then $\tau^*(P) = 1$. By definition, τ satisfies P, so P is satisfiable.

Definition 11.11 (Logical Implication)
Let $P, Q \in \mathcal{F}_{PV}$. P *logically implies* Q if and only if for any $\tau\colon PV \to \{0,1\}$, if τ satisfies P, then τ satisfies Q. Logical implication is denoted by $P \implies Q$.

Theorem 11.12 (Tautology and Logical Implication)
For any $P, Q \in \mathcal{F}_{PV}$, $P \implies Q$ if and only if $P \to Q$ is a tautology.

Proof: We have

$$\begin{aligned} P \implies Q &\text{ iff } \big(\text{if } \tau^*(P) = 1 \text{ then } \tau^*(Q) = 1\big) \text{ for any } \tau \\ &\text{iff } \tau^*(P \to Q) \text{ for any } \tau && \text{(by the definition of } \tau^*) \\ &\text{iff } P \to Q \text{ is a tautology} && \text{(by the definition of a tautology)} \end{aligned}$$

■

Definition 11.13 (Logical Equivalence)
Suppose that $P, Q \in \mathcal{F}_{PV}$. P is *logically equivalent* to Q if and only if for any truth assignment τ, τ satisfies P if and only if τ satisfies Q. Logical equivalence is denoted by P LEQV Q.

Theorem 11.14 (Tautology and Logical Equivalence)
For any $P, Q \in \mathcal{F}_{PV}$, P LEQV Q if and only if $P \leftrightarrow Q$ is a tautology.

PROOF: We have

$$\begin{aligned} P \text{ LEQV } Q &\text{ iff } \tau^*(P) = \tau^*(Q) \text{ for any } \tau \\ &\text{iff } \tau^*(P \leftrightarrow Q) = 1 \text{ for any } \tau \\ &\text{iff } \tau \text{ satisfies } P \leftrightarrow Q \text{ for any } \tau \end{aligned}$$

■

11.3 Normal Forms

Definition 11.15 (Literal)
A *literal* is a propositional formula consisting of only a variable or the negation of a variable.

Definition 11.16 (Term)
A *term* is a formula that is a literal or the conjunction of two or more literals.

Definition 11.17 (Clause)
A *clause* is a literal or the disjunction of two or more literals.

Definition 11.18 (Disjunctive Normal Form)
A propositional formula is in *disjunctive normal form (DNF)* if it is a term or the disjunction of two or more terms.

Definition 11.19 (Conjunctive Normal Form)
A propositional formula is in *conjunctive normal form (CNF)* if it is a clause or the conjunction of two or more clauses.

Example 11.20
Consider the formula $x \wedge y \wedge \neg z$. This is a both a CNF and a DNF; it is a CNF with three clauses, but also a DNF with one term.

Theorem 11.21 (DNF Theorem)
Every propositional formula is logically equivalent to a DNF formula.

PROOF: A sketch of a proof for a specific example is as follows. Let F be a formula with variables x, y, and z. Construct the truth table for F.

x	y	z	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Consider each truth assignment that satisfies F. The disjunction of all such truth assignments yields a formula that is logically equivalent to F. In this example, we have

$$F \text{ LEQV } (\neg x \wedge \neg y \wedge \neg z) \vee (\neg x \wedge y \wedge \neg z) \vee (\neg x \wedge y \wedge z) \vee (x \wedge \neg y \wedge z) \vee (x \wedge y \wedge \neg z)$$

This formula is in DNF. ■

Theorem 11.22 (CNF Theorem)
Every propositional formula is logically equivalent to a CNF formula.

PROOF: A sketch of a proof for a specific example is as follows. Let F be a formula with variables x, y, and z. Construct the truth table for $\neg F$.

x	y	z	F	$\neg F$
0	0	0	1	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	0	1

Consider each truth assignment that satisfies $\neg F$. We obtain a DNF formula equivalent to $\neg F$ by taking the disjunction of all such assignments; in this case, we get

$$\neg F \text{ LEQV } (\neg x \wedge \neg y \wedge z) \vee (x \wedge \neg y \wedge \neg z) \vee (x \wedge y \wedge z)$$

Now we obtain

$$\begin{aligned}
F &\text{ LEQV } \neg\neg F \\
&\text{ LEQV } \neg\big((\neg x \wedge \neg y \wedge z) \vee (x \wedge \neg y \wedge \neg z) \vee (x \wedge y \wedge z)\big) \\
&\text{ LEQV } \neg(\neg x \wedge \neg y \wedge z) \wedge \neg(x \wedge \neg y \wedge \neg z) \wedge \neg(x \wedge y \wedge z) && \text{(by De Morgan's law)} \\
&\text{ LEQV } (x \vee y \vee \neg z) \wedge (\neg x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z) && \text{(by De Morgan's law and the double negation law)}
\end{aligned}$$

This is a CNF formula equivalent to F. ■

The rest of the lecture was spent covering the "proving a set of connectives complete, and not complete" handout.

Week 12 Predicate Logic

12.1 Predicate Logic

Definition 12.1 (Predicate)
A *predicate* is a function

$$P\colon D^n \to \{0,1\}$$

where $D \neq \emptyset$ is the *domain* for the arguments and $n \in \mathbb{Z}^+$ is the *arity*. Alternatively, a predicate can be viewed as a relation

$$P \subseteq D^n$$

Definition 12.2 (First-Order Language)
The set of all predicate formulas is called a *first-order language (F-O language)*. It consists of

- An infinite set of variables (conventionally lowercase letters near the end of the alphabet)
- A set of predicate symbols, each with an associated arity (conventionally uppercase letters near the beginning of the alphabet)
- A set of constant symbols (conventionally lowercase letters near the beginning of the alphabet)

Definition 12.3 (Term)
A *term* is a variable or a constant.

Definition 12.4 (Atomic Formula)
An *atomic formula* is of the form

$$A(t_1, \ldots, t_n)$$

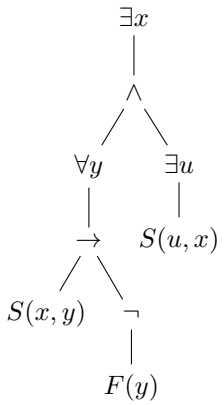
where A is a predicate with arity n and each t_i is a term.

Definition 12.5 (Set of F-O Formulas)
The *set of F-O formulas* is the smallest set such that:

Basis: Any atomic formula is in the set.

Induction Step: If F_1 and F_2 are in the set and x is a variable, then $\neg F_1$, $(F_1 \wedge F_2)$, $F_1 \vee F_2$, $(F_1 \to F_2)$, $(F_1 \leftrightarrow F_2)$, $\exists x\, F_1$, and $\forall x\, F_1$ are all in the set.

Example 12.6
Consider the F-O formula $\exists x \Big(\forall y \big(S(x,y) \to \neg F(y)\big) \wedge \exists u\, S(U,x)\Big)$. The *parse tree* of the formula is as follows:



The leaf nodes are atomic formulas, and the internal nodes are connectives and quantifiers with the variable they quantify.

Definition 12.7 (Free and Bound Variables)
Consider an occurrence of a variable x in a formula and the formula's parse tree. This occurrence of x in the formula is *free* if by tracing the path from the leaf node corresponding to x's occurrence back to the root node of the formula, $\exists x$ and $\forall x$ are not encountered.

Remark: *A rigorous structural induction definition of definition 12.7 can be found in section 6.2.3 of the course notes.*

Definition 12.8 (Sentence)
A formula with no free variables is called a *sentence*.

Example 12.9
Is the formula $\forall x\, P(x) \to P(c)$ satisfied?

Whether the formula is satisfied depends on:

- The domain
- The definitions of the predicates
- The values of the constants

Example 12.10
Is the formula $\forall x\, P(x) \to P(y)$ satisfied?

The formula may or may not be satisfied depending on:

- The domain
- The definitions of the predicates
- The values of the constants
- The values of the free variables

Definition 12.11 (Structure)
A *structure* of a F-O language consists of:

- The domain of the language, a non-empty set
- The definitions of the predicates
- The values of the constants

Definition 12.12 (Valuation)
A *valuation* of a F-O language consists of the values of the free variables.

Definition 12.13 (Interpretation)
A structure and a valuation together form an *interpretation*.

Remark: *An interpretation is to a predicate formula what a truth assignment is to a propositional formula.*

Example 12.14
Give an interpretation that satisfies $\forall x\, P(x) \to P(y)$.

Consider the domain D which is the set of all sandwiches, the predicate $P = \{x \in D : x \text{ contains meat}\}$, and let y be any sandwich. Since some sandwiches do not contain meat, $\forall x\, P(x)$ is falsified, so $\forall x\, P(x) \to P(y)$.

Definition 12.15 (Prenex Normal Form)
A predicate formula is in *prenex normal form (PNF)* if it is of the form

$$F = \mathbf{Q}_1 x_1 \mathbf{Q}_2 x_2 \dots \mathbf{Q}_k x_k E$$

where $k \in \mathbb{N}$, each $\mathbf{Q}_i \in \{\forall, \exists\}$, and E is quantifier-free.

Theorem 12.16
Every F-O formula has a logically equivalent formula in PNF.

The rest of the lecture was spent covering the "logical equivalences from course notes" handout. Some things to note about the handout:

- The quantifiers are denoted by $\mathbf{Q} \in \{\forall, \exists\}$
- $\overline{\mathbf{Q}} = \begin{cases} \exists & \text{if } \mathbf{Q} = \forall \\ \forall & \text{if } \mathbf{Q} = \exists \end{cases}$
- F_y^x is the formula F with all free occurrences of x replaced with y