CS 360
Lecture Notes
Spring 2024

Collin Roberts

June 28, 2024

Contents

1 Lecture 01

Outline

1. Introduction to CS 360 - Course Outline, M1 1-17
2. Terminology - M1 18-21
3. Languages - M1 22-28
4. Techniques of Proof - M1 29-39

1.1 Introduction to CS 360

1. Refer to the Course Outline and M1 slides 1-17 for the details.
2. I will lecture on the document camera, with the text and old lecture slides as resources.
3. I will announce pre-reading for all the future lectures by email.

1.2 Terminology

Definition 1.2.1. *An* ***alphabet*** *is a non-empty finite set of symbols, usually denoted* Σ.

Examples:

1. Binary alphabet: $\Sigma = \{0, 1\}$ Note, a surprising amount of work can be done over this small alphabet.
2. Latin alphabet: $\Sigma = \{a, b, \ldots, z\}$
3. Unary alphabet: $\Sigma = \{1\}$

Q: Why do we require that Σ be non-empty?
A:

1. If we allow $\Sigma = \emptyset$, then the only possible word over Σ is ε, the **empty word**, i.e. the word of no alphabet symbols.
2. The language $\{\varepsilon\}$ is not very interesting.
3. Further, we can already construct the empty word, ε, over **any** alphabet.

Definition 1.2.2. *A* ***string*** *(a.k.a.* ***word****) is an ordered sequence of alphabet symbols.*

Examples:

1. 1010110 (Binary)
2. *rover* (Latin)

Notation / Conventions:

1. Denote the **length** of the string x (i.e. the number of alphabet symbols composing x) by $|x|$.
2. Denote the **empty string** (i.e. the string of no symbols) by ε.

Remarks:

1. In CS 360, all strings have **finite lengths**.
2. By definition, $|\varepsilon| = 0$.
3. ε is never an alphabet symbol. ε indicates the absence of any alphabet symbols.
4. ε can be formed over **any** alphabet.

Q: Fix some finite non-empty Σ. Does the power set of Σ contain ε?

A: No. But it does contain $\emptyset$. E.g. let $\Sigma = \{0, 1\}$. Then $P(\Sigma) = \{\emptyset, \{0\}, \{1\}, \{0, 1\}\}$. Observe that $P(\Sigma)$ is always a subset of the languages over Σ.

Remember that ε and $\emptyset$ are two different things. ε is a word; $\emptyset$ is a language. The cardinality of $\{\varepsilon\}$ is 1; the cardinality of $\emptyset$ is 0.

Manipulating Strings

1. **Concatenation:** One string after another. (e.g. if $x = car$, $y = rot$, then $xy = carrot$.)

   Observations:

$$\begin{aligned}
\varepsilon\varepsilon &= \varepsilon \\
\varepsilon^k &= \varepsilon, \text{ for any } k \in \mathbb{N} \\
\varepsilon x &= x, \text{ for any } x \\
x\varepsilon &= x, \text{ for any } x
\end{aligned}$$

2. **Powers:** Let $k \in \mathbb{N}$. (Our convention in CS 360 will be that the first natural number is 0.) Let x be a string. Then x^k is k copies of x, concatenated together. (Convention: $x^0 = \varepsilon$, for any x.)
3. **Prefixes:** x is a **prefix** of y if $y = xz$, for some z. In other words a prefix of y is the first k characters of y, for some $k \geq 0$. **Remark:** ε is a prefix of every string.
4. **Suffixes:** x is a **suffix** of y if $y = zx$, for some z. In other words a suffix of y is the last k characters of y, for some $k \geq 0$. **Remark:** ε is a suffix of every string.
5. **Substrings:** x is a substring of y if $y = zxw$, for some z and w. A substring can be fully described
   (a) by its starting position and length, or

(b) by its starting and ending positions.

Remark: Every string x is a prefix, suffix and substring of itself.

6. **Counting characters:** $n_b(x) = \#$ of times the alphabet symbol b is found in x.

   Example: $n_b(banana) = 1$, $n_a(banana) = 3$.

   (a) **Q:** Is $n_\varepsilon(x)$ well-defined, for some (any) string x?
   A: No. Formally, ε is never included in any alphabet. Practically, there is no way to make this expression unambiguous. Asking for $n_\varepsilon(x)$ is analogous to asking how many factors of 1 can be pulled from some integer z.

   (b) **Q:** Can we extend this definition to $n_w(x)$, where w is a word?
   A: No. There are many problems associated with attempting to do this, e.g.

   i. Should $n_{aba}(ababa)$ be 1, or 2? Do occurrences of the word have to be non-overlapping?

   ii. We must at least enforce that the desired word be non-empty, for the above reason.

7. **Reversing a string:** $x^R = x$ written in reverse order.

   Example: if $x = stressed$, then $x^R = desserts$. **Not** x to the power of R!

8. **Palindromes:** x a **palindrome** if $x = x^R$, e.g. $x = radar$.

Remarks:

1. The length of a concatenation = sum of the individual lengths:

$$|xy| = |x| + |y|.$$

Questions and Answers:

1. Is ε an alphabet symbol?
   Answer: No. Every alphabet symbol must have length 1, but $|\varepsilon| = 0$.
2. Does every alphabet contain ε?
   Answer: No, as above.

1.3 Languages

Definition 1.3.1. *Let Σ be an alphabet. A* ***language over*** *Σ is any set of words constructed using symbols from Σ.*

Notation: Denote by Σ^* the set of **all** words constructed using symbols from Σ.

Examples of Languages:

1. The following sets are languages over $\Sigma = \{0, 1\}$:
   (a) $\{0, 1, 010, 001001, 0101\}$
   (b) $\{\varepsilon, 0, 00, 000, \ldots\}$
2. The following sets are languages over the Latin alphabet:
   (a) $\{car, rot, carrot, fish\}$

Questions and Answers:

1. Does every language contain ε?
   Answer: No. For example, two of the above three examples of languages do not contain ε.
2. Does a prefix of a string have to be over the same alphabet as the string itself?
   Answer: Yes.
3. Does a prefix of a string have to be a member of the same language as the string itself?
   Answer: No. The notions of prefix/suffix/substring of a given string, are completely separate from the question of to what language the given string might belong. For example, consider the above language $L = \{car, rot, carrot, fish\}$. Then $fish \in L$, however fi is a prefix of $fish$, and clearly $fi \notin L$.
4. Can two languages be equal?
   Answer: Yes. In fact, much of our work in CS 360 will be to prove that two languages, L_1, L_2, described in seemingly different ways, are equal (equal as sets, i.e. they contain exactly the same words).
5. Does the finiteness of alphabets and strings imply that all languages must be finite?
   Answer: No. For example, the language $\{\varepsilon, 0, 00, 000, \ldots\}$ is infinite.

Remarks:

1. **Why we insist that Σ is non-empty:** If $\Sigma = \emptyset$, then the only possible languages over Σ are $\emptyset$ and $\{\varepsilon\}$. There is not much to study here! **Q:** is $\emptyset = \{\varepsilon\}$? **A:** No. These sets have different cardinalities.
2. Since Σ is non-empty, therefore Σ^* is always infinite. If $a \in \Sigma$, then Σ^* contains the infinite subset $\{\varepsilon, a, aa, aaa, \ldots\}$, hence it must be infinite.
3. By definition, any language L over Σ is a subset of Σ^*.
4. A language can be finite or infinite.
5. We can describe a language by listing its elements if it is finite, or in any other unambiguous way if it is infinite.

Definition 1.3.2. *A* ***class*** *of languages is a set of languages.*

Questions and Answers:

1. Does a class have to contain only distinct languages?
   Answer: Yes. A class is a set after all. When we define a set, we are only interested in which elements belong to that set, regardless of order or possible repetitions.

Examples:

1.

$$\begin{aligned}\mathcal{C} \;=\; & \{\{1, 11, 111, 1111, 11111, \ldots\},\\ & \{0, 00, 000, 0000, 00000, \ldots\},\\ & \{0, 1, 00, 11, 000, 111, \ldots\},\\ & \{\varepsilon\}\}\end{aligned}$$

Remarks:

1. We will always fix our alphabet Σ, before doing anything else. We can make Σ as large as we need in any particular example.
2. Be careful with your set theory notation. Make it clear to your reader what is what. See the lecture slides for common pitfalls.
   (a) Make sure you understand the difference between $\emptyset$ and $\{\varepsilon\}$!
   (b) A string is **not** a member of a class of languages.
      i. Strings are members of languages.
      ii. Languages are members of classes.
      iii. The only class that is also a language: $\emptyset$
   (c) $L = \{\varepsilon\}$ is a language.
   (d) The language L can be part of a class of languages (it is part of $\mathcal{C}$).
   (e) ε cannot be part of a class of languages. (It is a word.)
3. Since languages are just sets, we can take unions, intersections and complements of languages. To compute complements correctly, we need to be clear about the "big set" inside of which all languages over Σ live: it will be Σ^*. Then for any langauge L over Σ, we always have that the complement $L' = \Sigma^* \setminus L$ is another langauge over Σ.
4. **Concatenations:** Let L and M be languages. Then

   $$LM = \{xy | x \in L, y \in M\}.$$

   Examples:

(a) $L = \{love, true\}$, $M = \{\varepsilon, ly, r\}$
$LM = \{love, true, truely, lovely, lover, truer\}$

(b) Let $\Sigma = \{0, 1\}$. Let $L = \{0, 00, 000, \ldots\}$ and $M = \emptyset$. I claim that $LM = \emptyset$. Why? Recall that $LM = \{xy | x \in L, y \in M\}$. Since $M = \emptyset$, it is not possible to find a $y \in M$. Thus we cannot construct any concatenation xy of the required form to lie in LM.

(c) Let $\Sigma = \{0, 1\}$. Let $L = \{0, 00, 000, \ldots\}$ and $M = \{\varepsilon\}$. **Q:** What is LM this time? **A:** $LM = L$ (proved soon)

5. **Powers:** Let $k \in \mathbb{N}$. L^k is k copies of L concatenated together, as in 4 above.
   Remarks: For any L,
   (a) $L^0 = \{\varepsilon\}$.
   (b) $L^1 = L$.
   Examples:
   (a) If $L = \{sorta, kinda\}$, then
   $L^2 = \{sortasorta, kindakinda, sortakinda, kindasorta\}$
6. **Kleene star operator** (so named after S.C. Kleene):
   (a) $L^* = \bigcup_{i=0}^{\infty} L^i = L^0 \cup L^1 \cup L^2 \cup \cdots$.
   (b) $L^+ = \bigcup_{i=1}^{\infty} L^i = L^1 \cup L^2 \cup L^3 \cup \cdots$.
   Examples:
   (a) If $L = \{sorta, very\}$, then
      - $L^* = \{\varepsilon, sorta, very, sortavery, verysorta, \ldots\}$, and
      - $L^+ = \{sorta, very, sortavery, verysorta, \ldots\}$.

   Remarks:
   (a) In general, $L^+ = LL^*$.
   (b) The proof is left as an exercise.

Examples of Recursively Defined Objects

1. the natural numbers, $\mathbb{N}$ (base: $n = 0$, otherwise $n = s(m)$, for some natural number, m, where s is the successor function). Inducting on this recursive structure leads to POMI.
2. in CS 245, syntactically correct propositional formulas. Inducting on this structure requires POSI.
3. a word w over some alphabet Σ (base $w = \varepsilon$, otherwise $w = xa$, for some alphabet symbol a and some word x). We'll need this setup soon, when we define the **extended transition function** for a **deterministic finite automaton (DFA)**.

Questions and Answers

1. Why is this base case allowed, when ε is never part of any alphabet?

A: ε is a word over any alphabet. It is simply the word containing no letters from the chosen alphabet.

1.4 Techniques of Proof

Three important proof ideas:

1. In CS 360, we often need to prove that two languages, L_1 and L_2, are equal. Since languages are just sets, this means proving that the sets are equal. To prove that language L_1 equals language L_2, show $L_1 \subseteq L_2$ and $L_2 \subseteq L_1$.
   Example: Prove that $LM = L$ in the earlier example (where $M = \{\varepsilon\}$).
   Recall we must prove $LM \subseteq L$ and $L \subseteq LM$.
   Proof that $LM \subseteq L$:
   - Let $xy \in LM$ be arbitrary.
   - In other words, let $x \in L$ and $y \in M$ be arbitrary.
   - Because $M = \{\varepsilon\}$, therefore $y = \varepsilon$.
   - Then $xy = x\varepsilon = x \in L$.

   Proof that $L \subseteq LM$:
   - Let $x \in L$ be arbitrary.
   - Then $x = x\varepsilon \in LM$.
2. To prove a statement is false, one counterexample is enough.
   Example: Is it true that every natural number which is congruent to 0 modulo 3 is congruent to 0 modulo 6?
   Answer: No.
   Counterexample: The natural number 9 is congruent to 0 modulo 3, but is not congruent to 0 modulo 6. Remember: A **single case** that is not true proves a $\forall$-conjecture false.
3. To prove some statement about all words in some language L (over some Σ) is true, induct on the structure of w. This can be
   (a) induction on $|w|$, if we have nothing else to go on, or
   (b) induction on the structure of $w \in L$, depending on how L is defined.

2 Lecture 02

Outline

1. Deterministic Finite Automata - M2 1-13
2. Nondeterministic Finite Automata - M2 14-30

2.1 Deterministic Finite Automata

Definition 2.1.1. *A* ***deterministic finite automaton (DFA)*** *is a model of a computer consisting of five ingredients:*

1. *Σ: an alphabet (for input words)*
2. *Q: a finite set of computation states*
3. *$\delta : Q \times \Sigma \to Q$: a transition function*
   - **Important:** *In a DFA, there must be a transition defined for* ***every state*** *and for* ***every alphabet symbol****.*
4. *$q_0 \in Q$: a start state*
5. *$F \subseteq Q$: a set of accept (final) states*

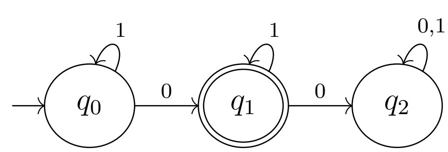
Intuition:

1. A DFA moves from one state to another as it reads each symbol in its input word.
2. At the end of the input, the machine either **accepts** or **rejects** the input, depending on whether the terminating state belongs to F, or not.

Vital constraints:

1. A DFA cannot work backward in its input; it can only go forward.
2. A DFA's only memory is its current state; aside from that, it has forgotten everything seen so far.

Example:



1. $\Sigma = \{0, 1\}$,
2. $Q = \{q_0, q_1, q_2\}$,
3. δ is defined as in the picture.
   It includes $(q_0, 0) \mapsto q_1$
4. initial/starting state $= q_0$,
5. $F = \{q_1\}$.

This DFA **accepts** the **language** of words over $\Sigma = \{0, 1\}$, having exactly one 0 symbol (i.e. $L(1^*01^*)$).

Question: How would you prove that?

Definition 2.1.2. *Let D be a DFA and let w be a word. We say that D* ***accepts*** *w if D terminates in an accept state after processing w.*

We need the next definition to make this precise.
Intuitively, we want $\hat{\delta}(q, w)$ to be the state we terminate in if we start at state q and follow δ for each letter in the word w in turn. To make this work for all words w, we must define the **extended transition function**, $\hat{\delta}$, recursively.

Definition 2.1.3. *$\hat{\delta}(q, w)$ is defined recursively, as follows:*

1. *$\hat{\delta}(q, \varepsilon) = q$, for all states q.*
2. *Otherwise, if $|w| > 0$, then we can write $w = xa$, for some alphabet symbol, a, and some word x over Σ. Then define $\hat{\delta}(q, w) = \delta(\hat{\delta}(q, x), a)$.*

Then we can make Definition 2.1.2 above precise by saying that D **accepts** w if $\hat{\delta}(q_0, w) \in F$.

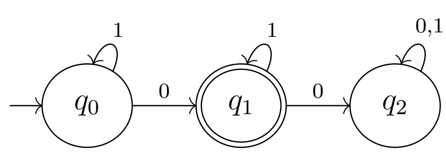
Lemma 2.1.4. *Let $D = (\Sigma, Q, q_0, F, \delta)$ be a DFA, with extended transition function $\hat{\delta}$. Let $q \in Q$ and $a \in \Sigma$ be arbitrary. Then $\hat{\delta}(q, a) = \delta(q, a)$, i.e. $\hat{\delta}$ agrees with δ for any state q and on any single alphabet symbol a.*

Proof.

$$\begin{aligned} \hat{\delta}(q, a) &= \hat{\delta}(q, \varepsilon a) \\ &= \delta(\hat{\delta}(q, \varepsilon), a) \\ &= \delta(q, a). \end{aligned}$$

□

Example: Letting M be our earlier DFA:



Show that M accepts 1011.

Solution: By the shape of M and by the refined Definition 2.1.2, we need to show that $\hat{\delta}(q_0, 1011) = q_1$. We compute

$$\begin{aligned}
\hat{\delta}(q_0, 1011) &= \delta(\hat{\delta}(q_0, 101), 1) && (1)\\
&= \delta(\delta(\hat{\delta}(q_0, 10), 1), 1) && (2)\\
&= \delta(\delta(\delta(\hat{\delta}(q_0, 1), 0), 1), 1) && (3)\\
&= \delta(\delta(\delta(\delta(\hat{\delta}(q_0, \varepsilon), 1), 0), 1), 1) && (4)\\
&= \delta(\delta(\delta(\delta(q_0, 1), 0), 1), 1) && (5)\\
&= \delta(\delta(\delta(q_0, 0), 1), 1) && (6)\\
&= \delta(\delta(q_1, 1), 1) && (7)\\
&= \delta(q_1, 1) && (8)\\
&= q_1 && (9)\\
&\in F. && (10)
\end{aligned}$$

Questions and Answers:

1. Are we not processing backwards here?
   A: No. Applying the definition of $\hat{\delta}$ is not processing anything. From line 5 onwards, we use δ repeatedly to process forward through the input word.

Definition 2.1.5. *Let $M = (Q, \delta, \Sigma, q_0, F)$ be a DFA, with extended transition function $\hat{\delta}$. The **language of** M**, denoted** $L(M)$ is the set of all words accepted by M:*

$$L(M) = \left\{ w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F \right\}.$$

Terminology: $L(M)$ can be called:

1. The language of the DFA M
2. The language **accepted** by the DFA M
3. The language **recognized** by the DFA M

Example: See lecture slides for an example of a DFA that accepts all binary strings which are the binary representation of a multiple of 3.

We are going to prove a theorem soon that FAs accept a specific class of languages, called **regular languages**.

- That should be robust: changes to an FA should not invalidate the property.
- So we will change FAs in a variety of small and large ways.
- The first major change is **nondeterminism**.

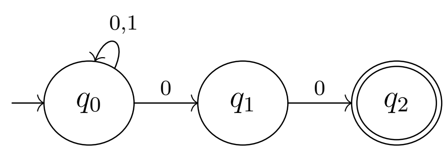
Questions and Answers:

1. Is the only difference between δ and $\hat{\delta}$ the types of their arguments?
   A: Yes.
   (a) Recall that $\delta : Q \times \Sigma \to Q$, i.e. it takes a (state, alphabet symbol) pair.
   (b) By contrast, $\hat{\delta} : Q \times \Sigma^* \to Q$, i.e. it takes a (state, word) pair.

2.2 Nondeterministic Finite Automata

- We enhance a DFA to create and NFA: given a state and an alphabet symbol, transition to some set of states (not necessarily a single state, as in a DFA).
- We will now need to keep track of multiple threads.

An Example: $L = \{$all words with 00 as the last two symbols$\}$



Questions and Answers:

1. Do we need to define outgoing transitions for the symbol 1 from states q_1 and q_2?
   Answer: No. See the remark immediately following the definition of δ for an NFA.

Definition 2.2.1. *NFA defined by 5 parameters*

1. Σ = *input alphabet*
2. Q = *finite set of computation states*
3. q_0 = *start state*
4. $F \subseteq Q$ = *accept states*
5. δ = *transition function*
   - **Important:** *In an NFA, there are 0,1, or multiple transitions defined for every (state, alphabet symbol) pair. $\delta(q, a)$ is a set of states, not a single state as in a DFA. If a thread reaches a state which has no outgoing transition for the given input symbol, then that thread* ***crashes****; it proceeds no further.*

Differences from DFAs:

1. $\delta : Q \times \Sigma \to \{$subsets of $Q\}$.

- Recall notation: $2^Q = \{\text{subsets of } Q\}$ = power set of Q.
- Based on a state in Q and an input letter from Σ, several states may now be now active in Q.
- (Different from DFA, where it is from $Q \times \Sigma \to Q$.)

2. $\hat{\delta}(q, w)$ = all states that we can reach from start state q processing the input word w.
3. $\hat{\delta}$: function $Q \times \Sigma^* \to 2^Q$.
4. The NFA accepts whenever **any** state path from q_0 terminates in an accept state.

Here is the definition for the extended transition function, $\hat{\delta}$, for an NFA.

Definition 2.2.2. *1. Base case:* $\hat{\delta}(q, \varepsilon) = \{q\}$.

2. Recursive case: If $|w| > 0$, *then we may write* $w = xa$, *where* $|a| = 1$.

- *Then define* $\hat{\delta}(q, xa) = \bigcup_{p \in \hat{\delta}(q,x)} \delta(p, a)$. *(defining* $\hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a)$ *doesn't work because* $\hat{\delta}(q, x)$ *is a set, not necessarily a single state.)*
  - ***Note:*** *This definition handles threads that crash correctly.*
    - *A thread that crashes has no outgoing transition for* (p, a).
    - *In other words,* $\delta(p, a) = \emptyset$.
    - *But then,* $\delta(p, a)$ *contributes nothing to* $\cup_{p \in \hat{\delta}(q,x)} \delta(p, a) = \hat{\delta}(q, xa)$, *reflecting the fact that the thread has crashed and proceeds no further.*

Definition of **acceptance** in an NFA:

Definition 2.2.3. *An NFA N* ***accepts*** *a word w if* $\hat{\delta}(q_0, w) \cap F \neq \emptyset$.

"When processing w, at least one thread terminates in an accept state."
Definition of the **language** of an NFA:

Definition 2.2.4.
- *The language of the NFA* $N = (\Sigma, Q, q_0, F, \delta)$ *is:*

$$L(N) = \left\{ w \in \Sigma^* | \hat{\delta}(q_0, w) \cap F \neq \emptyset \right\},$$

- *"all words accepted by the NFA".*

Theorem 2.2.5. *Let L be a language that is accepted by an NFA, N. Then L is accepted by some DFA.*

"NFAs are no more powerful than DFAs".

Outline of proof

1. Given an NFA N, construct a DFA, D, from N.
2. Then, show that $L(D) = L(N)$.

Remember:

1. The machine D does **not** have to have the same states as N, just the same alphabet and language!
2. See the motivating example in the slides of the **subset construction** we are about to generalize for our proof.

Proof. Let $N = (\Sigma, Q_N, q_0, F_N, \delta_N)$ be any NFA. Construct a new DFA $D = (\Sigma, Q_D, \{q_0\}, F_D, \delta_D)$, with these parameters:

- Q_D is the set of subsets of Q_N, which is denoted 2^{Q_N} in the lecture slides.
- Let D's initial state be $\{q_0\}$, where q_0 is the initial state of N. This is a legal state in Q_D (it is a particular subset of the states Q_N).
- $F_D = \{S \in Q_D \mid S \cap F_N \neq \emptyset\}$. (Recall by the definition of Q_D, any $S \in Q_D$ is some subset of the set Q_N of states of N.)
- and a more complicated transition function. For any $S \in Q_D$, and any $a \in \Sigma$, define:

$$\begin{aligned} & \delta_D(S, a) \\ = \; & \{\text{all states reachable in } N \text{ from } S \text{ when we read } a\} \\ = \; & \bigcup_{p \in S} \delta_N(p, a). \end{aligned}$$

Now we must show that $L(D) = L(N)$.

1. $w \in L(D)$ if and only if $\hat{\delta}_D(\{q_0\}, w) \in F_D$.
2. $w \in L(N)$ if and only if $\hat{\delta}_N(q_0, w) \cap F_N \neq \emptyset$. By the definition of F_D, 1 is equivalent to:
3. $w \in L(D)$ if and only if $\hat{\delta}_D(\{q_0\}, w) \cap F_N \neq \emptyset$.

Using 2 and 3, to show that $w \in L(D)$ if and only if $w \in L(N)$, it suffices to show that $\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$, for any word $w \in \Sigma^*$.

Proof: By induction on $|w|$.

1. Base case ($|w| = 0$): Thus $w = \varepsilon$.

- Then

$$
\begin{aligned}
& \hat{\delta}_D(\{q_0\}, \varepsilon) \\
\underbrace{=}_{D \text{ is a } DFA} \quad & \{q_0\} \\
\underbrace{=}_{N \text{ is an } NFA} \quad & \hat{\delta}_N(q_0, \varepsilon).
\end{aligned}
$$

2. Inductive case ($|w| > 0$):
   - The inductive hypothesis is that, for every $x \in \Sigma^*$ with $|x| < |w|$, we have $\hat{\delta}_D(\{q_0\}, x) = \hat{\delta}_N(q_0, x)$.
   - Since $|w| > 0$, we may write $w = xa$, for some $x \in \Sigma^*$ and $a \in \Sigma$.
   - Then the induction hypothesis applies to x.
   - We compute

$$
\begin{aligned}
\hat{\delta}_D(\{q_0\}, w) \quad & \underbrace{=}_{w=xa} \quad \hat{\delta}_D(\{q_0\}, xa) \\
& \underbrace{=}_{\text{Definition of } \hat{\delta}_D \text{ for the DFA } D} \quad \delta_D(\hat{\delta}_D(\{q_0\}, x), a) \\
& \underbrace{=}_{\text{induction hypothesis}} \quad \delta_D(\hat{\delta_N}(q_0, x), a) \\
& \underbrace{=}_{\text{Definition of } \delta_D} \quad \bigcup_{p \in \hat{\delta}_N(q_0, x)} \delta_N(p, a) \\
& \underbrace{=}_{\text{Definition of } \hat{\delta}_N} \quad \hat{\delta_N}(q_0, xa) \\
& \underbrace{=}_{w=xa} \quad \hat{\delta_N}(q_0, w).
\end{aligned}
$$

□

Remarks:

1. Nothing in the argument depended on any special properties of q_0. We could run the argument with **any** q. Thus we actually proved

$$\hat{\delta}\,(\{q\}, w) = \hat{\delta}(q, w),$$

for any $w \in \Sigma^*$.

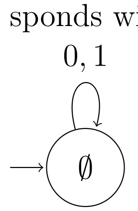
2. The opposite direction is easy: DFAs are NFAs! (Well, we must change δ trivially, so that the NFA transitions to a single state correctly reflect the original DFA definition.)
3. Although Theorems 2.2.5 and 3.1.7 will not appear on exams, ideas and techniques from these proofs are fair game.

3 Lecture 03

Outline

1. ε-Nondeterministic Finite Automata - M2 31-44

Q & A

1. Is it correct that, for some DFA, M, $\varepsilon \in L(M)$ if and only if M's initial state, q_0, is also a final state of M?
   A: Yes!
2. Why do we use the notation $2^Q = \{\text{all possible subsets of } Q\} = P(Q)$, the **power set** of Q?
   A: For intuition, start with the simplest case, in which Q is finite. Write $Q = \{q_1, \ldots, q_n\}$. Create a "flag" to indicate the inclusion (or not) for each q_i:
   $$\begin{cases} 1 & \text{if} \quad q_i \text{ is included} \\ 0 & \text{if} \quad q_i \text{ is excluded} \end{cases}$$
   Observe that every subset of Q corresponds 1-1 with an n-digit binary string. So there are $2^{|Q|}$ possible combinations.
3. When constructing D from N, can we get a crash state in D, violating the definition of a DFA?
   A: No. If all threads, from some subset of N's states crash, this corresponds with an arrow to
   

   in the DFA picture.

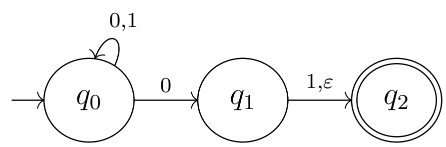
3.1 ε-Nondeterministic Finite Automata

- Sometimes in designing an NFA, it is handy to have transitions that happen automatically, without reading any letters of the input.

- Machine models that allow this are **ε-NFAs**.

Definition 3.1.1. *An ε-NFA is a 5-tuple, like an NFA. The only difference is transition function: δ is now a function: $Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$. (Transitions may exist that do not consume input letters.)*

Example: The ε-NFA



accepts binary words ending with 0 or with 01. (We could do this without ε-transitions, by making the second state an accept state.)

Defining the ε-closure of a state:

We want $\textsc{Eclose}(p)$ to be all states reachable starting from p only using ε-transitions. This motivates the following definition.

Definition 3.1.2. *Let E be an ε-NFA, having a state, p.*

- *Then define $\textsc{Eclose}(p)$ recursively:*
  1. *(Base) $p \in \textsc{Eclose}(p)$*
  2. *(Recursive) If $q \in \textsc{Eclose}(p)$, then so are all of the states in $\delta(q, \varepsilon)$.*

*The set $\textsc{Eclose}(p)$ is called the **ε-closure** of the state p.*

Defining the ε-closure of a set of states

Definition 3.1.3. *Let E be an ε-NFA, having $S \subseteq Q_N$. Then define $\textsc{Eclose}(S)$ via:*

$$\textsc{Eclose}(S) = \bigcup_{s \in S} \textsc{Eclose}(s).$$

Lemma 3.1.4. *For an ε-NFA E, having states Q, a subset $S \subseteq Q$ and a decomposition $S = \bigcup_i S_i$,*

$$\bigcup_i \textsc{Eclose}(S_i) = \textsc{Eclose}\left(\bigcup_i S_i\right),$$

(i.e. taking ε-closure commutes with taking set unions).

Proof.

$$\begin{aligned}
\bigcup_i \textsc{Eclose}(S_i) &\underbrace{=}_{\text{Definition of } \textsc{Eclose}(S_i)} \bigcup_i \left(\bigcup_{s \in S_i} \textsc{Eclose}(s) \right) \\
&\underbrace{=}_{S = \bigcup_i S_i} \bigcup_{s \in S} \textsc{Eclose}(s) \\
&\underbrace{=}_{\text{Definition of } \textsc{Eclose}(S)} \textsc{Eclose}(S) \\
&\underbrace{=}_{S = \bigcup_i S_i} \textsc{Eclose}\left(\bigcup_i S_i \right). \square
\end{aligned}$$

$\square$

Definition 3.1.5. *We define $\hat{\delta}$ for ε-NFA's, also recursively.*

1. *Base case: $\hat{\delta}(q, \varepsilon) = \textsc{Eclose}(q)$:*
   i.e. states reachable from q via ε-transitions alone
2. *Inductive case: Suppose that $w = xa$, where $a \in \Sigma$.*
   *(**Note**: a **cannot be** ε, which is not a member of Σ.)*
   (a) *Let $P = \hat{\delta}(q, x)$, i.e. P is the set of all states in Q that we can get to by following either edges for the letters of x or ε-transitions (including ε-transitions at the end of x).*
   (b) *Then, we must follow the transitions for the alphabet symbol a: Let $R = \bigcup_{p \in P} \delta(p, a)$: then R has all of the states we can get to from P after following a transition for a.*
   (c) *Last, we might have some more ε-transitions.*
   (d) *So, define $\hat{\delta}(q, w) = \textsc{Eclose}(R) = \textsc{Eclose}\left(\bigcup_{p \in \hat{\delta}(q,x)} \delta(p, a) \right)$*

Definition 3.1.6.

- *Language of an ε-NFA:*
  $L = \left\{ w \in \Sigma^* \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset \right\}$
- *That is, $\hat{\delta}(q_0, x)$ includes an accept state.*

Q: Are ε-NFAs more powerful than DFAs?

A: No.

Theorem 3.1.7. *Given an ε-NFA E, there exists an ordinary DFA D such that $L(D) = L(E)$.*

Remarks:

1. This is not very surprising: we must show that we can include the ε-transitions of E in the transition function δ_D for D.
2. (The other direction is just by definition: a DFA is an ε-NFA, once we make some trivial changes to the structure of δ so that it produces a 1-element set when it reads in a symbol, and the empty set when it reads in ε.)
3. To prove the theorem, we must construct a DFA, D, accepting language $L(E)$.
   (a) Write $E = (\Sigma, Q_E, q_0, F_E, \delta_E)$.
   (b) We will construct $D = (\Sigma, Q_D, q_0, F_D, \delta_D)$.

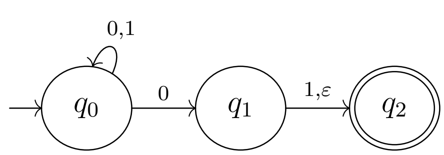
Proof. • Both machines use the same alphabet, of course.

- We use the subset construction, as when we built the DFA for an NFA.
  1. $Q_D = \{$all possible subsets of states from $Q_E\}$
  2. $F_D = \{S \in Q_D \mid S \cap F_E \neq \emptyset\}$
  3. The starting state is $q_D = \textsc{Eclose}(q_0)$. We start having implicitly taken ε-transitions from the starting state q_0 of E.
  4. Transition function. δ_D:
     - From one DFA state, S (which is a set of states in E), if we process one letter, a, in the new DFA, we should mimic this behaviour from E:
       * follow any edges labelled a, and
       * take any ε-transitions
     - From any one state from E, say p, this then takes us to:
       * $\delta_E(p, a)$, and then to
       * $\textsc{Eclose}(\delta_E(p, a))$
  5. And we therefore want the union over all states $p \in S$:

$$\delta_D(S, a) = \bigcup_{p \in S} \textsc{Eclose}(\delta_E(p, a)).$$

Subset Construction Example:

- Here we demonstrate one step in the subset construction for the earlier small example of an ε-NFA:



- The subset construction gives us

$$\begin{aligned} Q_D &= \{\emptyset, \{q_0\}, \{q_1\}, \{q_2\}, \{q_0, q_1\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\}\} \\ q_D &= \text{ECLOSE}(\{q_0\}) = \{q_0\} \end{aligned}$$

- Now we determine $\delta_D(S, a)$, where $S = \{q_0, q_1, q_2\}$ (the state we reach from $\{q_0\}$ upon reading 0) and $a = 1$.
- Computing $\text{ECLOSE}(\delta_E(p, a))$ for each $p \in S$ gives

$$\begin{aligned} \text{ECLOSE}(\delta_E(q_0, 1)) &= \text{ECLOSE}(q_0) = \{q_0\} \\ \text{ECLOSE}(\delta_E(q_1, 1)) &= \text{ECLOSE}(\{q_2\}) = \{q_2\} \\ \text{ECLOSE}(\delta_E(q_2, 1)) &= \text{ECLOSE}(\emptyset) = \emptyset \end{aligned}$$

- Hence the target state coming from the subset construction is $\{q_0, q_2\}$.
- The construction says that we need to add to the transition function for D: $\delta_D(\{q_0, q_1, q_2\}, 1) = \{q_0, q_2\}$.
- Now to complete the transition function for D, we do this same construction for each of the 8 choices for S (above) and each alphabet symbol from $\Sigma = \{0, 1\}$.

Now, to show equality of the languages of the two automata, we must show that if x is accepted by E, then x is accepted by D, and vice versa.

- (One particular concern: do we do the right thing for the word ε?)

Goal: Prove that $w \in L(E)$ if and only if $w \in L(D)$.

1. $w \in L(E)$ if and only if $\hat{\delta}_E(q_0, w) \cap F_E \neq \emptyset$.
2. $w \in L(D)$ if and only if $\hat{\delta}_D(\text{ECLOSE}(q_0), w) \in F_D$. Using the definition of F_D, rewrite 2 as
3. $w \in L(D)$ if and only if $\hat{\delta}_D(\text{ECLOSE}(q_0), w) \cap F_E \neq \emptyset$.

Now using 1 and 3, to show that $w \in L(E)$ if and only if $w \in L(D)$, it suffices to prove that $\hat{\delta}_E(q_0, w) = \hat{\delta}_D(\text{ECLOSE}(q_0), w)$, for every word $w \in \Sigma^*$.

The proof is by induction on $|w|$.

1. Base case ($|w| = 0$): In this case, $w = \varepsilon$. We have $\hat{\delta}_E(q_0, \varepsilon) = \text{ECLOSE}(\{q_0\})$, by definition of $\hat{\delta}_E$ in the ε-NFA.

- We therefore have

$$
\begin{aligned}
\hat{\delta}_E(q_0, \varepsilon) \quad & \underbrace{=}_{\varepsilon-NFA \text{ rule}} \quad \textsc{Eclose}(\{q_0\}) \\
& \underbrace{=}_{\text{singleton set property}} \quad \textsc{Eclose}(q_0) \\
& \underbrace{=}_{\text{Definition of } q_D} \quad q_D \\
& \underbrace{=}_{D \text{ is a } DFA} \quad \hat{\delta}_D(q_D, \varepsilon) \\
& \underbrace{=}_{\text{Definition of } q_D} \quad \hat{\delta}_D(\textsc{Eclose}(q_0), \varepsilon).
\end{aligned}
$$

- So the base case holds.

2. <u>Inductive case ($|w| > 0$):</u>
   - The induction hypothesis, for every $x \in \Sigma^*$ with $|x| < |w|$, we have that $\hat{\delta}_E(q_0, x) = \hat{\delta}_D(\textsc{Eclose}(q_0), x)$.
   - Write $w = xa$, for some $x \in \Sigma^*$ and $a \in \Sigma$.
   - The induction hypothesis applies to x.
   - For notational convenience let $S = \hat{\delta}_E(q_0, x) = \hat{\delta}_D(\textsc{Eclose}(q_0), x)$.

- Now we compute

$$
\begin{aligned}
\hat{\delta}_E(q_0, w) &\underbrace{=}_{w=xa} \hat{\delta}_E(q_0, xa) \\
&\underbrace{=}_{\text{Definition of } \hat{\delta}_E} \text{ECLOSE}\left(\bigcup_{p \in \hat{\delta}_E(q_0, x)} \delta_E(p, a)\right) \\
&\underbrace{=}_{\text{Definition of } S} \text{ECLOSE}\left(\bigcup_{p \in S} \delta_E(p, a)\right) \\
&\underbrace{=}_{\text{Lemma 3.1.4}} \bigcup_{p \in S} \text{ECLOSE}(\delta_E(p, a)) \\
&\underbrace{=}_{\text{Definition of } \delta_D} \delta_D(S, a) \\
&\underbrace{=}_{\text{Definition of } S} \delta_D(\hat{\delta}_D(\text{ECLOSE}(q_0), x), a) \\
&\underbrace{=}_{\text{Definition of } \hat{\delta}_D} \hat{\delta}_D(\text{ECLOSE}(q_0), xa) \\
&\underbrace{=}_{\text{w=xa}} \hat{\delta}_D(\text{ECLOSE}(q_0), w), \text{ as required.}
\end{aligned}
$$

□

Remarks:

1. You should convince yourself that the base case handles the input word ε correctly.
2. This argument did not use any special properties of q_0.
3. We could re-run the argument with any state, q.
4. Thus we could have proved, for any state q and any word $w \in \Sigma^*$:

$$\hat{\delta}_E(q, w) = \hat{\delta}_D(\text{ECLOSE}(q), x).$$

4 Lecture 04

Outline

1. Regular Expressions - M3 1-5
2. Regular Languages - M3 6-12

4.1 Regular Expressions

Definition 4.1.1. *Let Σ be an alphabet. We construct the* ***regular expressions*** *over Σ (and describe the* ***language*** *which each regular expression represents, i.e. the set of words that fit the mold defined by the regular expression) recursively, as follows.*

Base

1. *$\emptyset$ is a regular expression, and $L(\emptyset) = \emptyset$.*
2. *ε is a regular expression, and $L(\varepsilon) = \{\varepsilon\}$.*
3. *If $a \in \Sigma$ is any symbol, then a is a regular expression and $L(a) = \{a\}$.*

Induction

1. *If E and F are regular expressions, then $E + F$ is a regular expression, and $L(E + F) = L(E) \cup L(F)$.*
2. *If E and F are regular expressions, then EF is a regular expression, and $L(EF) = L(E)L(F)$.*
3. *If E is a regular expression, then E^* is a regular expression, and $L(E^*) = (L(E))^*$.*
4. *If E is a regular expression, then (E) is a regular expression, and $L((E)) = L(E)$.*

Remarks:

1. For any word $w \in \Sigma^*$, the regular expression for $L = \{w\}$ is just w itself.
2. As in algebra, there is an order of operations here:
   (a) **Parentheses** are used to override (or emphasize) the default order as needed.
   (b) *
   (c) **Concatenation**
   (d) $+$

Examples: Let $\Sigma = \{0, 1\}$.

1. Let $L(0^*1^*) = \{$ strings starting with some number of 0s and ending with some number of 1s $\}$.
2. Let $L(01^*) = \{$ strings starting with exactly one 0 and ending with some number of 1s $\}$.
3. Let $L((01)^*) = \{$ zero or more copies of 01 $\}$.

Adding parentheses makes concatenation come before $*$ in 3, as opposed to 2. Hence the languages in 2 and 3 must be different.

4.2 Regular Languages

Definition 4.2.1. *A language L is* ***regular*** *if it obeys the following recursive definition:*

Base

1. $L = \emptyset$ is regular.
2. $L = \{\varepsilon\}$ is regular.
3. $L = \{a\}$ for some alphabet letter $a \in \Sigma$ is regular.

Induction

1. $L = L_1 \cup L_2$, for regular languages L_1, L_2
2. $L = L_1 L_2$ for regular languages L_1, L_2
3. $L = L_1^*$, for some regular language L_1

No other languages are regular.

Remarks:

1. It might appear at first glance that the union part of the regular language definition says that every language is regular (every language is a possibly infinite union of 1-word languages, each of which is regular, by Theorem 4.2.2, below). But infinite unions are not the same as finite unions.
2. Non-regular languages exist e.g. $\{0^i 1^i \mid i \geq 0\}$. In L06 we will develop a technique to establish the non-regularity of a language (Pumping Lemma for Regular Languages).
3. Remember that $\emptyset \neq \{\varepsilon\}$!

Theorem 4.2.2. *Every one-word language is regular.*

Proof. See Lecture Slides. □

Theorem 4.2.3. *Every finite language is regular.*

Proof. See Lecture Slides. □

Remarks:

1. The converse of Theorem 4.2.3 is false! For example $L(0^*)$ is regular, and this is always infinite.

Examples: (trim down for class; refer to slides for the rest)

1. If $\Sigma = \{0, 1\}$, then $\Sigma^* = \{0, 1\}^*$.
   Regular expression: $(0 + 1)^*$

2. Even-length sequences:
   Can be divided into 2-letter sub-words. $(00 + 11 + 01 + 10)^*$ or $((0 + 1)(0 + 1))^*$
3. Sequences with length at most 3: $\varepsilon + (0 + 1) + (0 + 1)(0 + 1) + (0 + 1)(0 + 1)(0 + 1)$ or
   $(0 + 1 + \varepsilon)(0 + 1 + \varepsilon)(0 + 1 + \varepsilon)$ or
   $\varepsilon + 1 + 0 + 11 + 10 + 01 + 00 + 111 + \cdots$
4. Sequences with at most two zeros:
   $1^*(0 + \varepsilon)1^*(0 + \varepsilon)1^*$
5. And lots more.

Rules For Regular Languages - NOT Exhaustive (trim down for class; refer to slides for the rest): Basic rules that you can often use (not exciting, but true...):

1. Let E be a regular expression. Then $\emptyset E = E\emptyset = \emptyset$ (If $w \in L(\emptyset E), w = xy$ where $x \in L(\emptyset)$ and $y \in L(E)$. But nothing works for x.)
2. $\emptyset^* = \{\varepsilon\}$ (might be surprising). Recall, $L^0 = \{\varepsilon\}$, for any L. Thus $\emptyset^0 = \{\varepsilon\}, \emptyset^i = \emptyset$, for every $i \geq 1$, as above.
3. $\{\varepsilon\}^* = \{\varepsilon\}$
4. $x + x = x$ (remember, + means union)
5. $(x^*)^* = x^*$ (Taking closure twice, or by induction any finite number of times, equals taking closure once)
   - Side Note: an operation for which applying the operation twice equals applying the operation once is called **idempotent**.
   - Other natural examples of idempotents are projections in linear algebra.
6. $x(y + z) = xy + xz$

Tons more of these, but we will not focus on them.
The first two might be surprising, and are thus important.

5 Lecture 05

Outline

1. Kleene's Theorem - M3 13-39

5.1 Kleene's Theorem

Kleene's Theorem 5.1.1. *Every regular language is the language of a DFA, and every DFA accepts a regular language.*

Remarks about Theorem 5.1.1, Before Its Proof:

1. Recall: We saw that DFAs are equally powerful to ε-NFAs.
2. Given any regular language, we will construct and ε-NFA which accepts it.
3. We will prove that the language of any DFA is regular.
4. Together these constructions will establish the Theorem.

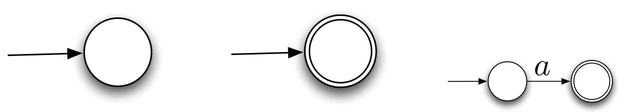
Forward **To Prove:** Let L be any regular language. I claim that there is an ε-NFA M, such that $L(M) = L$. We will show an ε-NFA for the base cases (easy), then prove the other three cases by structural induction on L (more interesting).

Q & A

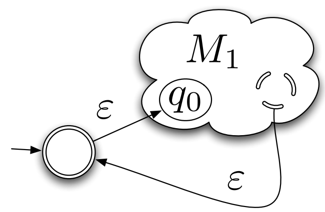
1. Why are all three base cases needed?
   A: We are inducting on the structure of the regular language, L. Therefore every base case and every inductive case from the construction of L must be included in our proof.

Base Cases:

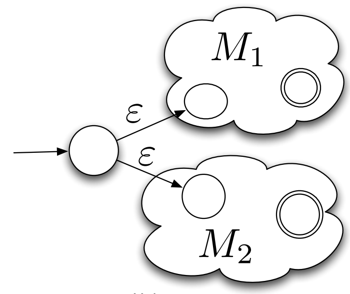
$\emptyset$ $\quad\quad$ $\{\varepsilon\}$ $\quad\quad$ $\{a\}$



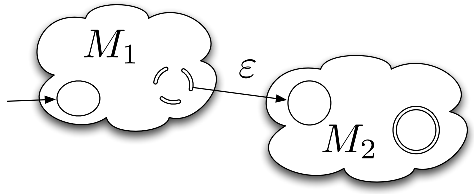
Inductive Cases:



Kleene Star



Union



Concatenation

Here is an argument that this works, for the Kleene * operator, where $L = (L_1)^*$. By results from Module 2, let M_1 be an ε-NFA which accepts L_1.

<u>Proof that $L(M) \subseteq L$:</u>

- Let $x \in L(M)$ be arbitrary.
- Then there is a path in M for x ending in the start state.
- This path can then be broken down into 0 or more sub-paths, each of which begins and ends in the initial state.
- Break the full path into its individual loops.
- Each constitutes a path, in M_1, from the initial state to some accept state.
- In other words, each loop is taken by some $x_1 \in L(M_1)$.
- Write $x = x_1 \cdots x_k$, for some $k \geq 0$.
- Therefore x is a concatenation of zero or more words in L_1, in other words $x \in L_1^*$. (Note that x can be ε.)
- This shows that $L(M) \subseteq (L_1)^* = L$.

<u>Proof that $L \subseteq L(M)$:</u>

- Let $x \in L = L_1^*$ be arbitrary.
- Write $x = x_1 x_2 x_3 \cdots x_k$, with all $x_i \in L_1$.
- So there is a path from start state in the new machine to an accept state for each of the x_i.
- Join the paths together, following each with the ε-transition back to the start state to get a path for x from the start state back to itself.
- The start state is an accept state, so our new machine accepts x.

- So $x \in L(M)$.
- This shows that $L \subseteq L(M)$.

Now we are done.

- We have shown that $L(M) \subseteq L$ and $L \subseteq L(M)$.
- Therefore we have $L(M) = L$, as claimed.

The other inductive cases are similar and are left as exercises.

<u>Backward</u> **To Prove:** For every DFA, its language is regular.

One idea: Given a DFA D, we must find a regular expression for its language. Paths through a state can be replaced by the regular expression that represents going from the previous state to the next one. This leads us to the idea of **state removal**. See the Lecture Slides for details. However we will not write our proof this way here.

Recall:

- Let $D = \{\Sigma, Q, q_0, F, \delta\}$ be an arbitrary DFA.
- Then $L(D) = \{x \in \Sigma^* \mid \hat{\delta}(q_0, x) \in F\} = \bigcup_{r \in F}\{x \in \Sigma^* \mid \hat{\delta}(q_0, x) = r\}$.
- **Idea:** Characterize all input strings that take us from state q to state r: Define $L(q, r) = \{x \in \Sigma^* \mid \hat{\delta}(q, x) = r\}$, for any $q, r \in Q$.
- With this notation, $L(D) = \bigcup_{r \in F} L(q_0, r)$ (i.e. the set of words that take us from q_0 to any accept state $r \in F$)
- This is a finite union, so if all $L(q_0, r)$ are regular, then so is $L(D)$. (Prove this, by induction on $|F|$!)
- Although the union is finite, some (all) of the sets involved can be infinite.

What we now want:

Theorem 5.1.2. *Let $D = (\Sigma, Q, q_0, F, \delta)$ be an arbitrary DFA. Let $q, r \in Q$ be arbitrary. Define $L(q, r) = \{x \in \Sigma^* \mid \hat{\delta}(q, x) = r\}$. Then $L(q, r)$ is regular.*

Remarks on the Proof:

- We will prove this using structural induction, after giving an appropriate setup.
- Think about the list of the states visited while we process some $x \in L(q, r)$.
- Restrict the number of states that can come between q and r, and grow this set of states.
- Number the states of the DFA $D:\ 1, \ldots, n$.
- For $0 \le k \le n$, define $L(q, r, k) = \{$ all words in $L(q, r)$ where all **intermediate** states between q and r are from $0, \ldots, k\}$.
  (Remember, M is a DFA, so each word has only one state path.)

- The $k = 0$ case says that **no** intermediate states can exist, i.e. q and r are equal, or there is a single transition from q to r.

Remarks on the languages $L(q, r, k)$

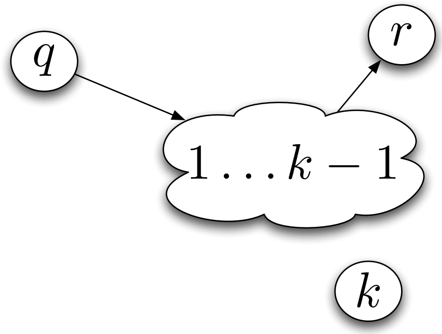
- Reducing k shrinks the set; increasing k grows the set.
- If the path from q to r for word x uses state k, then x is not in $L(q, r, k-1)$.
- We induct on $0 \leq k \leq n$.
- This will give us what we want: $L(q, r, n) = L(q, r)$.
- (Different labellings of states will produce different $L(q, r, k)$ sets, but the proof will work regardless.)
- Formally: For $0 \leq k \leq n$, $L(q, r, k) = \{x \in \Sigma^* \mid \hat{\delta}(q, x) = r$ and $\hat{\delta}(q, w) \leq k$ for all non-empty proper prefixes of w of x (i.e. where $w \neq \varepsilon$ and $w \neq x)\}$.
- (Terminology: w is a **proper** prefix of x if w is a prefix of x and not equal to x.)
- New Goal: Prove $L(q, r, n)$ is regular, for all q, r.
- It is enough show that $L(q, r, k)$ is regular for all $0 \leq k \leq n$.
- The proof is by induction on k.

Proof. Base case: $k = 0$:

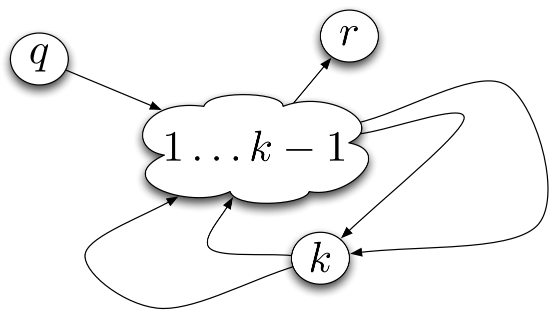
- If $x \in L(q, r, 0)$, then $\hat{\delta}(q, w) \leq 0$ for all proper prefixes w of x.
- But that means there are no proper prefixes w of x.
- Therefore, $|x| = 0$ or 1.
- So all words in $L(q, r, 0)$ are of length 0 or 1.
- As Σ is finite, this implies $L(q, r, 0)$ is finite, and thus regular by Theorem 4.2.3.

Induction step: $k > 0$:

- Inductive hypothesis: $L(q, r, k-1)$ is regular for all q and r.
- We must show that $L(q, r, k)$ is regular.
  - Let $x \in L(q, r, k)$ be arbitrary.
  - If the path from q to r for x never touches the state k, then $x \in L(q, r, k-1)$.



- Otherwise: $x \in L(q, r, k)$ and $x \notin L(q, r, k-1)$, i.e. k *is* on the path from q to r for x, one or more times.
- Then there is a first and a last time we are in state k. Between the first and last time, all states are $\leq k$, regardless of whether $q > k$ or $r > k$ (by the definition of $x \in L(q, r, k)$).



So we can divide x into:

1. The part from q to k the first time,
2. The "middle part", from k back to itself 0 or more times:
   (a) The first loop (if any) from k to k,
   (b) The next loop from k to k,
   (c) . . .
   (d) The last loop from k to k,
3. and the part from k to r.

Words divided in this way are exactly the words in $L(q, r, k)$ that include state k on their state path. This set of words is $L(q, k, k-1)(L(k, k, k-1))^* L(k, r, k-1)$.

So the language $L(q, r, k)$ is the union of two languages:

1. $L(q, r, k-1)$, which we know is regular by the induction hypothesis, and

2. $\underbrace{L(q,k,k-1)}_{1}\underbrace{(L(k,k,k-1))^*}_{2}\underbrace{L(k,r,k-1)}_{3}$, which is the concatenation of three languages (each of which is regular by the induction hypothesis and the definition of regular languages), and thus is also regular.

Hence $L(q,r,k)$ is regular. We are done, but that may not be obvious yet.

- We proved that $L(q,r,k)$ is regular for all k.
- We noted that $L(q,r) = L(q,r,n)$, so $L(q,r)$ is always regular for any $q,r \in Q$.
- Therefore $L(q_0,r)$ is regular for any $r \in F$.
- Thus $\bigcup_{r\in F} L(q_0,r)$ is regular (as it is the union of a finite number of regular languages).
- But $L(D) = \bigcup_{r\in F} L(q_0,r)$ is therefore regular.

□

So $L(D)$ is regular! (Again, how long is the expression for $L(q_0,r)$?)
This is the end of the proof of Kleene's Theorem:

- Given a DFA, we have shown that its language is regular.
- Given a regular language, we can produce an ε-NFA which recognizes it.
- NFAs and ε-NFAs have the same computing power as DFAs.

Next module: the boundaries of regular languages, and closure rules for them.

6 Lecture 06

Outline

1. Pumping Lemma for Regular Languages - M4 1-20

6.1 Pumping Lemma for Regular Languages

Goal: Develop a rigourous way of proving that some language, L, **is not regular**.

Non-regular languages By Kleene's Theorem, a language L is regular if and only if there exists a DFA, M, such that $L = L(M)$. How does a DFA M work?

- Suppose M has n states, for some $n > 0$.
- Consider a word $x \in L(M)$ with $|x| \geq n$.

- On its path from q_0 to an accept state, x must repeat a state somewhere along the path.
- **Reason:** There are only n states in total, and the machine starts out in one of them (namely q_0), then reads $\geq n$ input characters. (Recall the **pigeonhole principle**.)
- Let's say that we repeat state r.
- Then the word x can be decomposed: $x = uvw$, where:
  - u = the part from q_0 to the first time we reach r (i.e. after processing u, we are in state r).
  - v = the first loop from r to itself (i.e. after processing v, we are again in state r).
  - w = The part from the second time we reach r that leads us to an accept state.
  - **Note:** it is possible that either u or w is ε, but v cannot be ε.
  - We are in a DFA, without ε-transitions, hence we must consume at least one input symbol to loop from r back to itself.
  - All we had to assume to obtain this decomposition was that $|x| \geq n$, where $n = \#$ of states in our DFA.
- This decomposition is possible for **any** word $x \in L(M)$ with $|x| \geq n$.
- **Fact**: $uvvw$ is also in $L(M)$. Why?
  - vv also loops from r back to itself:

$$\hat{\delta}(r, vv) \underbrace{=}_{\text{A01}} \hat{\delta}(\hat{\delta}(r, v), v) = \hat{\delta}(r, v) = r.$$

- <u>Another Fact:</u> $uw = uv^0w \in L(M)$.
- By induction, $uv^iw \in L(M)$, for all $i \in \mathbb{N}$.
- If we choose the **first** time a state is repeated, then $|uv| \leq n$.
  - **Reason:** The machine has n states, so we must have the first repeated state by the n^{th} step.)
- We can "pump out" many copies of v, for example $uvvvvvvvvvvvw$ is also part of $L(M)$.

Pumping Lemma For Regular Languages 6.1.1. *For every regular language L, there* ***exists*** *some positive integer n such that all words $x \in L$ with $|x| \geq n$ can be decomposed as $x = uvw$, where:*
- *$|uv| \leq n$,*
- *$v \neq \varepsilon$, and*
- *$uv^iw \in L$ for all $i \geq 1$.*

Proof. Already done! □

Remarks:

1. Think of n as being the number of states in some (smallest) DFA, M, accepting L.
2. Lemma 6.1.1 describes one feature of all long words in a regular language:
   (a) For some definition of "long", **all long words can be pumped**.
   (b) Note that, if L is finite (and therefore regular), then taking any $n > \max_{x \in L}\{|x|\}$ works (because with such an n, L contains **no** long words).

This provides us with the following **Technique for Proving that a Language Cannot Be Regular**

Let L be a language. Suppose that for **any** positive integer, n:

1. There exists a word $x \in L$ with $|x| \geq n$ such that
2. for **any** decomposition of x into $x = uvw$, with $|uv| \leq n$ and $v \neq \varepsilon$,
3. $uv^*w \not\subseteq L$.

Then L is **not** a regular language.

The Same Technique, Explained in English

"Suppose that for any value of $n > 0$,

1. there exists a word $x \in L$ with $|x| \geq n$"... *(If there is always a long word in L)*
2. "such that for any decomposition of x into $x = uvw$, with $|uv| \leq n$ and $v \neq \varepsilon$" ... *(that cannot be decomposed into three parts where the first 2 parts are not long and the middle part is non-trivial)*
3. "$uv^*w \not\subseteq L$."... *(and the second part cannot be pumped,)*

Then L is not a regular language.

Examples of Proving Languages Are Non-Regular

Theorem 6.1.2. $L = \{0^i1^i | i \geq 0\} = \{\varepsilon, 01, 0011, 000111, 00001111, \ldots\}$ *is not a regular language.*

Proof.
- Let $n > 0$ be arbitrary.
- Choose a word $x \in L$, with $|x| \geq n$. (N.B. there is no recipe for choosing this x.)
- I will choose $x = 0^n1^n$. This is our **long word**.
- Now, consider all possible decompositions $x = uvw$, where $|uv| \leq n$, and $v \neq \varepsilon$.

- **Fact**: for all such decompositions, $uv = 0^k$ for some $0 < k \leq n$, because the first n characters of $x = uvw$ are all 0 (by the definition of x).
- Now, we must argue that $uv^*w \not\subset L$ is not a subset of L. In particular, we must exhibit some $i \geq 0$ such that $uv^iw \notin L$. (often, $i = 0$ or $i = 2$ will work.)
- Let $i = 0$. Recall that v is all 0's. Then uv^0w will have fewer 0's than 1's. So $uv^0w \notin L$.
- And hence the language L is not regular.

□

Again, how did that work?

Pumping lemma: to prove languages are not regular.

- For any definition of **long**, find a long word in the language:
  Long: length $\geq n$. Our long word was $x = 0^n1^n$.
- Consider all breakdowns of x into $x = uvw$, where uv is short and $v \neq \varepsilon$.
  For the long word x, if $x = uvw$, and uv is short, then uv is all 0's.
- If for all of these breakdowns $x = uvw$, we cannot pump v, then L is not regular.
  No matter what v is, it must be all 0's. So if we pump v, then $uvvw$ or uw both have the wrong number of 0's. So L is not regular.
- We can also prove L is not regular by thinking of possible DFAs for L and showing that they cannot exist.
- This is hard in general. The Pumping Lemma is better.

Another example

Theorem 6.1.3. *The language $L = \{0^p | p$ is a prime $\}$ is not regular.*

Why can we not pump the primes?

Proof.
- (This language includes $00, 000, 00000, 0000000, 00000000000, \ldots$)
- Proof by Pumping Lemma. (Assume that there are infinitely many primes. There are many nice proofs of this fact.)
  - Choose a value of $n > 0$.
  - Choose $x = 0^p$, for a prime $p \geq n$.
  - Then x is a long word in L.
  - Now we must argue that no decomposition of x can be pumped.

  Consider all decompositions $x = uvw$, where $|uv| \leq n$ and $v \neq \varepsilon$.
  - Then $v = 0^k$ for some $1 \leq k \leq n$.

- And $uv^*w = \{0^{p-k}, 0^p, 0^{p+k}, 0^{p+2k}, \ldots\}$.
  - Is it possible that all of these are in L?
  - No. One member of uv^*w is $0^{p+(pk)}$; it is the $(p+2)^{th}$ member in the above list.
  - This word is not a member of L, since $p+pk = (1+k)p$ is composite (both factors are non-trivial, as $k \geq 1$).
- For any n, we can find a long word, such that all decompositions of it cannot be pumped. Therefore L is not regular.

□

Another example: palindromes

Theorem 6.1.4. *$L = \{s \mid s = s^R\}$, the language of palindromes, is not regular.*

Remarks:

1. Examples of palindromes: $0110, 01110, \varepsilon, 1111$, etc.

Proof. Proof by Pumping Lemma.

- Given a value of $n > 0$, find a word in L of length at least n.
- How about $x = 0^n10^n$?
- Now, consider all decompositions of this into $x = uvw$, where uv is short and v is not ε.
- Again, v must be 0^i for some $1 \leq i \leq n$.
- And the number of 0's before the only 1 in uv^2w is more than the number after it, so it cannot be a palindrome.
- So we cannot pump x, regardless of our choice of decomposition.
- So L is not regular.

□

One more example

Theorem 6.1.5. *Let $L = \{y!z \mid |y| > |z|, y, z \in \{0,1\}^*\}$, over $\Sigma = \{0, 1, !\}$. Then L is not regular.*

Remarks:

1. This language includes words like $111!00, 1!, 10001!111$.

Proof. Proof by Pumping Lemma.

- Consider a value $n > 0$.
- The string $x = 0^n!0^{n-1}$ is long, and in L.

- We will show that uv^0w is not in the language.
  - Decompose $x = uvw$ with uv of length at most n and nonempty v.
  - For all such decompositions, $v = 0^k$ for some $k \geq 1$.
  - And $uv^0w = 0^{n-k}!0^{n-1}$.
  - This is not a word in L: the part before the ! character is too short.
  - So v is not pumpable, no matter how we do it.
- L is not regular.

$\square$

What can go wrong?

It is easy to misuse the Pumping Lemma.

- The existence of one bad (i.e. non-pumpable) decomposition of x does not matter.
- We must show that **all** decompositions of $x = uvw$ with $|uv| \leq n$ and $v \neq \varepsilon$ cannot be pumped.

Example:

- Obviously, $L = (01)^*$ is regular.
- For any value of $n \geq 2$, $(01)^n$ is a long word in L.
- Decompose into $u = 0, v = 1, w = (01)^{n-1}$.
- Then $|uv| \leq n$ and $v \neq \varepsilon$.
- Also $uv^2w = 011(01)^{n-1} \notin L$.
- So we conclude that L is not regular (?!?!?)

Clearly we have done something wrong! What is it?

- Problem: We must show that **no** decomposition can be pumped. One bad (non-pumpable) decomposition is not enough.
- The decomposition $u = \varepsilon, v = 01, w = (01)^{n-1}$ is pumpable.

More Pumping Lemma: pitfalls

- The Pumping Lemma:
  - Long words in regular languages can be pumped.
- Its contrapositive:
  - If a language has long words that cannot be pumped, it is not regular.
- **Note**: the theorem does not give a definition of regular languages. The following is **not** true:
  - If all long words in a language can be pumped, it is regular.
- In fact, some non-regular languages can be pumped.

7 Lecture 07

Outline

1. Closure Rules for Regular Languages - M4 21-28
2. Decision Problems for Regular Languages - M4 29-36

7.1 Closure Rules for Regular Languages

Definition 7.1.1. 1. *A class of languages is **closed** under a binary operation if applying that operation to two languages in the class always yields a language in the class.*
2. *A class of languages is **closed** under a unary operation if applying that operation to one language in the class always yields a language in the class.*

Theorem 7.1.2. *Regular languages are closed under $*$, **finite** union and **finite** concatenation.*

Proof. Obvious, by Definition 4.2.1 plus induction. □

Remarks:

1. Subsets of regular languages are **not necessarily regular**: If $\Sigma = \{0,1\}$, then $(0+1)^* = \Sigma^*$ is regular, so any language over $\Sigma = \{0,1\}$ is the subset of a regular language!
2. $\{0^i 1^i \mid i \geq 0\} \subseteq (0+1)^*$ but is not regular.
3. **Moral:** The class of regular languages is NOT closed under subset.

Theorem 7.1.3. *If language L is regular, then so is its complement, L'.*

Proof. We will apply Kleene's Theorem 5.1.1. Given a regular L, we will construct a DFA accepting the complement L'.

- Since L is regular, by Kleene's Theorem, L is the language of some DFA, M, with state set Q and accept states $F \subseteq Q$.
- Construct a new DFA, M' from M, by swapping the accept and reject states in M, in the specification of M'.
- Then M', with accept set $Q \setminus F$ accepts all words which the old DFA, M, rejected and rejects all words which M accepted.
- Because M is a DFA, therefore so is M'.
- By construction, M' accepts L'.
- By Kleene's Theorem 5.1.1, L' is regular.

□

Remarks:

1. This is much easier than constructing a regular expression for L' from a regular expression for L.
2. The Theorem says that regular languages are closed under complements.

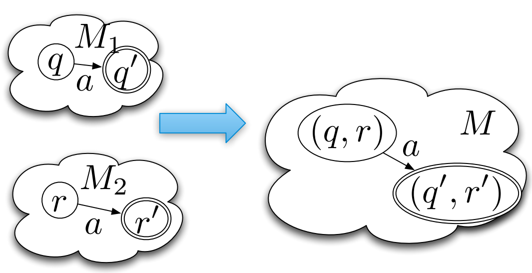
Theorem 7.1.4. *If L_1 and L_2 are regular languages, then so is $L_1 \cap L_2$.*

Remarks:

1. In other words, regular languages are closed under intersection.
2. This can be proved in lots of correct ways.
3. One nice way: $L_1 \cap L_2 = (L_1' \cup L_2')'$. (Draw a suitable Venn diagram to see the equality.)
4. L_1' is regular (by Theorem 7.1.3); so is L_2'.
5. By Definition 4.2.1, $L_1' \cup L_2'$ is regular.
6. And again, by our Theorem 7.1.3, $(L_1' \cup L_2')' = L_1 \cap L_2$ is regular.

Proof. **Strategy: by building a DFA** By Kleene's Theorem, let M_1 and M_2 be DFAs accepting L_1 and L_2 respectively. We must construct a new DFA, M, such that $L(M) = L_1 \cap L_2$. **Idea:** M simulates processing each alphabet symbol, $a \in \Sigma$, through M_1 and M_2 in parallel. M accepts exactly when both M_1 and M_2 accept simultaneously. Define M using the ingredients

- Σ is the same as for M_1, M_2.
- $Q = \{(q_1, q_2) \mid q_1 \in Q_1, q_2 \in Q_2\}$
- $F = \{(f_1, f_2) \mid f_1 \in F_1, f_2 \in F_2\}$
- $q_0 = (q_{1_0}, q_{2_0})$
- $\delta : \delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$
- By construction, M is a DFA, which accepts $L_1 \cap L_2$.
- By Kleene's Theorem, $L(M) = L_1 \cap L_2$ is regular.



□

Closure under reversal Recall: w^R is the **reversal** of the word w.
Given a language L, let L^R be the language that consists of all of the words of L, reversed.

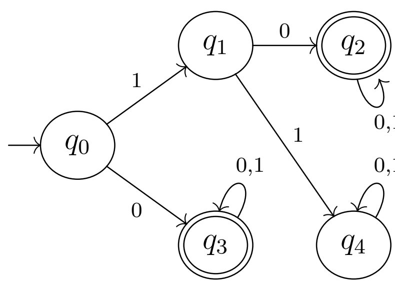
Theorem 7.1.5. *If L is regular, then so is L^R.*

Proof idea (we won't make it rigourous): Let M be a finite automaton whose language is L. Make a new finite automaton R with the same states as M, plus a new start state:

- All of the edges of R are the reversals of the edges of M.
- The sole accept state of R is the start state of M.
- The start state of R has an ε-transition to each accept state of M.

Then R reverses the automaton M: if we start at an accept state of M and work our way back to the start state of M (i.e. if M accepted x), then the new machine R accepts the word x^R, and vice versa.

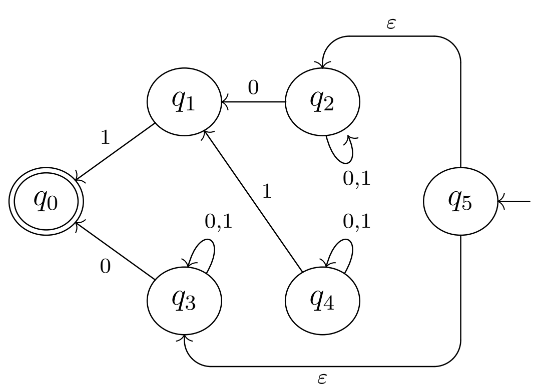
Closure under reversal Suppose we have the following DFA, M



to accept the language

$$L = \{w \mid w \text{ begins with 0 or with 10}\}.$$

Closure under reversal The construction yields this ε-NFA, R



to accept the language

$$L = \{w \mid w \text{ ends with 0 or with 01}\}.$$

Note that, although this ε-NFA R can be simplified, the construction is still correct.

Proof. We can prove this theorem by structural induction on the construction of the regular language L.

- Base cases: If $L = \emptyset, \{\varepsilon\}$ or $\{a\}$,then $L^R = L$ is regular.
- Induction cases:
  - If $L = L_1 \cup L_2$ for regular languages L_1 and L_2, then $L^R = L_1^R \cup L_2^R$, and both of L_1^R and L_2^R are regular by induction.
  - If $L = L_1 L_2$ for regular languages L_1 and L_2, then $L^R = L_2^R L_1^R$, and this is the concatenation of the regular languages L_2^R and L_1^R (by induction), and hence is regular.
  - If $L = L_1^*$, then
    * We want to prove that $L^R = (L_1^*)^R$ is regular.
    * The induction hypothesis is that L_1^R is regular.
    * Therefore by the definition of regular languages, it follows that $(L_1^R)^*$ is regular.
    * I claim that $(L_1^*)^R = (L_1^R)^*$. Proving the claim will complete this case.
    * For an arbitrary word w, we have

$$\begin{aligned} & w \in (L_1^*)^R \\ \Leftrightarrow\quad & w = (w_1 \cdots w_n)^R, \text{ for some } w_i s \in L_1 \\ \Leftrightarrow\quad & w = w_n^R \cdots w_1^R, \text{ for some } w_i s \in L_1 \\ \Leftrightarrow\quad & w \in (L_1^R)^*, \text{ which establishes the claim.} \end{aligned}$$

□

Remarks:

1. **Moral:** The class of regular languages is closed under reversals.
2. **Summary:** The class of regular languages is closed under:
   (a) finite concatenations, finite unions, $*$-closure
   (b) complements
   (c) complements
   (d) intersections

   The class of regular languages is NOT closed under taking subsets.

7.2 Decision Problems for Regular Languages

Recall that a **decision problem** is a question that requires a yes/no answer, given some input, e.g.

1. Given an arbitrary propositional formula, is it satisfiable? (decidable).
2. Given an arbitrary program `P` and an arbitrary input `I` for `P`, does `P` halt when processing `I`? (undecidable). This ia the **Halting Problem**. We will revisit the Halting Problem during the third chunk of CS 360, using the computation model of **Turing machines**.

Algorithmic questions about finite automata We do not often create algorithms in this class (apart from constructing automata, which model computations). Here is one exception:
Is it possible to find algorithms for the following:

1. Given a DFA M and a word x, does M accept x?
2. Given a DFA M, is $L(M) = \emptyset$?
3. Given a DFA M, is $L(M)$ infinite?
4. Given two DFAs M_1 and M_2, is $L(M_1) \cap L(M_2)$ empty?
5. Given two DFAs M_1 and M_2, is $L(M_1) \subseteq L(M_2)$?
6. Given two DFAs M_1 and M_2, is $L(M_1) = L(M_2)$?
7. Given two regular expressions e_1 and e_2, do they generate the same language?

1. Given a DFA M and a word x, does M accept x?
   - Just simulate the DFA.
   - M is a DFA, and the input x is of finite length, so M terminates after finitely many steps of processing x.
   - When M terminates, we determine whether M accepts x, by checking whether the terminating state is final or not.
   - This works because M is a DFA; If M is a Turing machine, then it might never terminate when processing x (Halting problem again).
2. Given a DFA M, is $L(M) = \emptyset$?
   More fun.

Lemma 7.2.1. *If a DFA, M, having n states accepts any words, then it must accept a word with $< n$ letters.*

Remark: The Lemma provides a finite set which we can examine to obtain our answer. Without the Lemma, we would need to check every member of the infinite set Σ^*.

Proof.
- Assume $L(M) \neq \emptyset$.

- By Kleene's Theorem, $L(M)$ is regular.
- Let $x_0 \in L(M)$ be arbitrary.
- If $|x_0| < n$, then we are finished.
- Otherwise, $|x_0| \geq n$ and by the proof of the Pumping Lemma, we can decompose $x_0 = u_0v_0w_0$, with $u_0w_0 \in L(M)$ and $|v_0| \geq 1$.
- We have produced a new word $u_0w_0 \in L(M)$, with $|u_0w_0| < |u_0v_0w_0|$.
- If $|u_0w_0| < n$, then we are finished.
- Otherwise, $|u_0w_0| \geq n$ and $u_0w_0 \in L(M)$ and so by the proof of the Pumping Lemma, we can decompose $u_0w_0 = u_1v_1w_1$, with $u_1w_1 \in L(M)$ and $|v_1| \geq 1$.
- Continuing in this way we obtain a sequence of words in $L(M)$ having strictly decreasing lengths: $x_0, u_0w_0, u_1w_1, \ldots, u_jw_j, \ldots$.
- As x_0 has finite length, after at most $|x_0| - n + 1$ steps, we will obtain a word in $L(M)$ with length $< n$. $\square$

$\square$

Now, back to the decision problem.
- Given a DFA M, is $L(M) = \emptyset$?
  - Try **every** word of length less than n (finitely many since our alphabet is finite).
  - If **any** word is accepted, then $L(M) \neq \emptyset$. Otherwise $L(M) = \emptyset$.
  - If no short word is accepted, then by Lemma 7.2.1, $L(M) = \emptyset$.

Next decision problem:
- Given a DFA M, is $L(M)$ infinite?

Theorem 7.2.2. *If M is a DFA with n states, then $L(M)$ is infinite if and only if $L(M)$ includes a word x satisfying $n \leq |x| < 2n$.*

Proof.
- Suppose $x \in L(M)$ and $n \leq |x| < 2n$.
- From the Pumping Lemma, x must be pumpable.
- The word $x = uvw$ can be used to generate the infinite language uv^*w, which is a subset of $L(M)$.
- So $L(M)$ is infinite.

Other direction: Assume that $L(M)$ is infinite.
- For a contradiction, suppose that there does not exist any $x \in L(M)$ satisfying $n \leq |x| < 2n$.
- In other words, every word $x \in L(M)$ with length at least n must have length at least $2n$.

- Let $x \in L(M)$ be a shortest word with length at least n (so that $|x| \geq 2n$ by the above point).
- (If there is no $x \in L(M)$ with length at least n, then $L(M)$ is finite, which cannot happen.)
- Decompose $x = uvw$, where $v \neq \varepsilon$ and $|uv| \leq n$, so that $1 \leq |v| \leq n$.
- By the Pumping Lemma, uw is also in $L(M)$.
- We have only removed at most n letters by removing v, so $|uw| \geq n$, and by construction, $|uw| < |x|$.
- Thus uw violates the choice of x as a shortest word in $L(M)$ with length at least n.
- This contradiction completes the proof.

$\square$

- If $L(M)$ is infinite, we must have a word in $L(M)$ with length between n and $2n$.
- To see if $L(M)$ is infinite, check all words between n and $2n$ in length.
- This runs in a finite amount of time.

Disjoint languages, subset

- Given two DFAs M_1 and M_2, is $L(M_1) \cap L(M_2)$ empty?
  - First, construct a DFA for $L(M_1) \cap L(M_2)$.
  - Then use the algorithm for testing for an empty language of an FA to see if it accepts the empty language.
- Given two DFAs M_1 and M_2, is $L(M_1) \subseteq L(M_2)$?
  - If so, then $L(M_2)' \cap L(M_1)$ is empty.
    (There is nothing in $L(M_1)$ that is not in $L(M_2)$.)
  - Build the DFA for $L(M_2)'$.
  - Use it to build the DFA for $L(M_2)' \cap L(M_1)$.
  - Use the algorithm from before to test if its language is empty!

Two FAs with the same language

- Given two DFAs M_1 and M_2, is $L(M_1) = L(M_2)$?
  - Yes, if $L(M_1) \subseteq L(M_2)$ and $L(M_2) \subseteq L(M_1)$.
  - Use the algorithm for testing for subset twice.
- Given two regular expressions e_1 and e_2, do they represent the same language?
  - Construct the DFAs for each regular expression, using Kleene's Theorem.
  - Then use the algorithm for testing if two FAs have the same language.

(If you like this kind of stuff, take CS 462.)

8 Lecture 08

Outline

1. Context-Free Grammars and Languages - M5 1-25

8.1 Context-Free Grammars and Languages

Definition 8.1.1. *a* ***context-free grammar*** *(CFG) is a 4-tuple: $G = (V, T, P, S)$, where*

1. *V is a finite set of* ***variables****, usually denoted by capital letters.*
2. *T is a finite alphabet, called* ***terminals****. $T \cap V = \emptyset$.*
   T is the alphabet for the CFG's ***language****.*
3. *P is a finite set of* ***productions****, which are definitions for the variables. Each production in P is of the form $A \to \alpha$, where*
   - *A is a variable, and*
   - *α is a string of symbols from V and T. (Therefore $\alpha \in (V \cup T)^*$), OR α may equal ε.*
   - ***Shorthand Notation****: If we have productions $A \to \alpha$ and $A \to \beta$, shorthand this as $A \to \alpha \mid \beta$.*
4. *S =* ***Start variable****: the symbol from V where the derivation of a word begins.*

An Example, which generates $L(0^*1^*)$ (Exercise: prove it!)
$G = (V, T, P, S)$, where

- $V = \{A, B, S\}$
- $T = \{0, 1\}$
- $P = \{S \to AB, A \to 0A \mid \varepsilon, B \to B1 \mid \varepsilon\}$
- $S = S$

Goal: Define what it means for a word to be in the **language of a CFG**.

Definition 8.1.2. *We write $\alpha \underset{G}{\Rightarrow} \beta$ if β can be produced from α via a* ***single*** *derivation in G.*

E.g., with the above G, $00AB1 \underset{G}{\Rightarrow} 00AB11$.

Definition 8.1.3.

- *Suppose that there is a sequence of strings $\gamma_1, \gamma_2, \ldots, \gamma_k$ for some $k \geq 1$ such that:*

1. $\alpha = \gamma_1$*,*
2. $\beta = \gamma_k$*,*
3. For all $2 \leq i \leq k$*,* $\gamma_{i-1} \underset{G}{\Rightarrow} \gamma_i$*.*

- *Then write* $\alpha \underset{G}{\overset{*}{\Rightarrow}} \beta$*.*

Example: Using the above G,

$$S \Rightarrow AB \Rightarrow 0AB \Rightarrow 00AB \Rightarrow 00\varepsilon B \Rightarrow 00B1 \Rightarrow 00\varepsilon 1.$$

This derivation says that $S \underset{G}{\overset{*}{\Rightarrow}} 001$.

Remarks on the Example:

1. We know the derivation is finished when the string contains only terminals, no variables.
2. This derivation uses the **leftmost rule** (the leftmost variable must be resolved first before moving on to other variables).
3. Our convention will be to use leftmost derivations.

Language of a grammar

Definition (Language of a CFG) 8.1.4. *Given a context-free grammar* $G = (V, T, P, S)$*, its **language** is:*

$$L(G) = \left\{ w \in T^* \mid S \underset{G}{\overset{*}{\Rightarrow}} w \right\}.$$

In English, $L(G)$ *is the set of words that we can derive from the start variable* S*, with no variables from* V *remaining.*
A language, L*, is **context free** if* $L = L(G)$*, for some CFG,* G*.*

A Bit More About the Leftmost Rule: Let $G : S \to AB, A \to 0A \mid \varepsilon, B \to 01$. Then we have

$$S \Rightarrow AB \Rightarrow \varepsilon B \Rightarrow \varepsilon 01, \text{ so } S \underset{G}{\overset{*}{\Rightarrow}} 01$$

We cannot produce 01 using the other rule for A first. The leftmost rule forces us to resolve A, somehow, before we resolve B. Note that the leftmost rule does not dictate **which** production for A we choose.

Why do we call these languages "context-free"?

- It does not matter **where** (i.e. in what context) a variable occurs.
- Suppose P includes: $A \to ab$.

- Given any string x_1Ax_2, we can change the A into ab.
- That is, we can do this substitution **regardless of the context** of A, which is what x_1 and x_2 are.
- Think about mad-libs from when you were a kid; they were funny because you were making sentences that were silly. They did not have context.
- Proper English (or indeed, any human language) has context.

An example $G = (V, T, P, S)$, where
- $V = \{S\}$,
- $T = \{0, 1\}$,
- $P = \{S \to 0S1 \mid S1 \mid \varepsilon\}$, and
- $S = S$.

Remark: We do not need to specify V, T or S when they are obvious, given P.
- $G : S \to 0S1 \mid S1 \mid \varepsilon$ is enough here.

The language of this CFG Let $L = \{0^i1^j$, where $0 \leq i \leq j\}$
- Proof sketch, using the Pumping Lemma that L is not regular:
  - Let $n > 0$ be arbitrary.
  - Let $x = 0^n1^n \in L$ be our long word.
  - Write $x = uvw$, where $|uv| \leq n$ and $v \neq \varepsilon$.
  - Then $v = 0^k$ for some $1 \leq k \leq n$, so $uv^2w \notin L$.

Q & A
1. Given some CFL of interest, possibly described in set theory notation, e.g. $\{0^i1^i \mid i \geq 0\}$, how does one produce a CFG, G, such that $L(G) = G$?
   A: There is no algorithm for choosing such a G. Human intuition is required to do this. We develop our intuition by working with CFGs and their languages. One additional complication - such a G need not be unique!

Theorem 8.1.5. *Let $G : S \to 0S1 \mid S1 \mid \varepsilon$. Let $L = \{0^i1^j$, where $i \leq j\}$. Then $L(G) = L$.*

Proof. To prove the Theorem, we must show $L \subseteq L(G)$ and $L(G) \subseteq L$.
Proof that $L \subseteq L(G)$: Let $x \in L$ be arbitrary. We will prove, by induction on $|x|$, that $x \in L(G)$.
- Base case: $|x| = 0$. Then $x = \varepsilon$, and $S \to \varepsilon$ is a rule in the grammar G. Therefore we have $S \overset{*}{\Rightarrow} \varepsilon$, so $\varepsilon \in L(G)$. So the base case holds.

- Inductive case: Let $|x| > 0$, so that $x \neq \varepsilon$. Write $x = 0^i 1^j$, for some $i \leq j$.
  - The induction hypothesis is that, for any $w \in L$, with $|w| < |x|$, $w \in L(G)$ (i.e. $S \overset{*}{\underset{G}{\Rightarrow}} w$).
  - We have these two cases for i:
    1. $\underline{i = 0}$, so $x = 1^j = 1^{j-1}1$ ($j \geq 1$ as we are not in the base case).
       * Then $1^{j-1} \in L$ by the definition of L.
       * The induction hypothesis applies to 1^{j-1}.
       * Thus $S \overset{*}{\Rightarrow} 1^{j-1}$.
       * Derive x in G via

$$S \Rightarrow S1 \overset{*}{\Rightarrow} (1^{j-1})1 = 1^j = x.$$

       * This shows that $x \in L(G)$ in this case.
    2. $\underline{i > 0}$, so that $x = 0^i 1^j$, with $0 < i \leq j$.
       * Then $0^{i-1}1^{j-1} \in L$ ($i \leq j$ implies $i - 1 \leq j - 1$).
       * The induction hypothesis applies to $0^{i-1}1^{j-1}$.
       * So $S \overset{*}{\Rightarrow} 0^{i-1}1^{j-1}$.
       * Derive x in G via

$$S \Rightarrow 0S1 \overset{*}{\Rightarrow} 0(0^{i-1}1^{j-1})1 = 0^i 1^j = x.$$

       * This shows that $x \in L(G)$ in this case.

    In either case, $x \in L(G)$. This finishes the proof that $L \subseteq L(G)$.

Proof that $L(G) \subseteq L$: Let $x \in L(G)$ be arbitrary. The proof is by induction on k, the number of steps in the derivation of x ($S \overset{k}{\Rightarrow} x$).

- Base case ($k = 1$; $k = 0$ is not possible: $V \cap T = \emptyset$): The only 1-step derivation in G that produces a word is $S \to \varepsilon$. Therefore $x = \varepsilon$. Because $\varepsilon = 0^0 1^0$, therefore $x \in L$. So the base case holds.
- Inductive case ($k > 1$): I.H.: every word $w \in L(G)$ derived in $< k$ steps is in L.
  - As we are not in the base case, the first step in the derivation of x must be $S \to 0S1$ or $S \to S1$.
  - If the first step is $S \to 0S1$,
    * Write $x = 0w1$ where $S \overset{*}{\Rightarrow} w$ in $< k$ steps.
    * By the induction hypothesis, $w \in L$.
    * Write $w = 0^i 1^j$, for some $i \leq j$.

    - Then $x = 0w1 = 0(0^i 1^j)1 = 0^{i+1}1^{j+1} \in L$.
  - If the first step is $S \to S1$,
    - Write $x = w1$ where $S \overset{*}{\Rightarrow} w$ in $< k$ steps.
    - By the induction hypothesis, $w \in L$.
    - Write $w = 0^i 1^j$, for some $i \leq j$ ($i \leq j$ implies $i \leq j + 1$).
    - Then $x = w1 = (0^i 1^j)1 = 0^i 1^{j+1} \in L$.
- In either case, $x \in L$.
- This finishes the proof that $L(G) \subseteq L$.

Put them together: $L(G) \subseteq L$ and $L \subseteq L(G)$. So $L = L(G)$ as claimed. □

This take a bit of care to set up, but theorems like this one are not hard to prove.

Remarks:

1. We described our L in two different ways:
   (a) set-theoretic
   (b) as the language of some CFG, G.

   Then we rigourously proved that the descriptions are interchangeable, i.e. that the two languages are equal.

Another Example of a CFL

Theorem 8.1.6. *Let $L = \{w \in \{0,1\}^* \mid n_0(w) = n_1(w)\}$.*

- *L consists of all binary strings with as many 0s as 1s.*

Then L is context free.

Proof. We will show that L is the language of the grammar: $G = (V, T, P, S)$, where

- $V = \{S\}$,
- $T = \{0, 1\}$,
- $P = \{S \to \varepsilon \mid 0S1 \mid 1S0 \mid SS\}$, and
- $S = S$.

To prove this, we must show that the languages are subsets of each other, in both directions.

Proof that $L(G) \subseteq L$:

- Let $w \in L(G)$ be arbitrary, i.e. assume $S \overset{n}{\Rightarrow} w$, for some n.
- The proof is by induction on n.
- Base case ($n = 1$): Then $w = \varepsilon$ (the only one-step derivation in G is $S \to \varepsilon$). Note that $\varepsilon \in L$, so the base case holds.

- Inductive case ($n \geq 2$): The induction hypothesis is that for every $x \in L(G)$ which is derived in fewer than n steps, we have $x \in L$.

Inductive step in one direction Consider the first step in a derivation of w. We have these cases (which are exhaustive, as we are no longer in the base case):

- $\underline{S \to 0S1}$: Write $w = 0x1$, where $S \overset{n-1}{\Rightarrow} x$. The induction hypothesis applies to x, so $n_0(x) = n_1(x)$. Hence, by construction, $n_0(w) = n_1(w)$ too. Thus $w \in L$.
- $\underline{S \to 1S0}$: Write $w = 1x0$, where $S \overset{n-1}{\Rightarrow} x$. The induction hypothesis applies to x, so $n_0(x) = n_1(x)$. Hence, by construction, $n_0(w) = n_1(w)$ too. Thus $w \in L$.
- $\underline{S \to SS}$: Write $w = xy$, where $S \overset{*}{\Rightarrow} x$ and $S \overset{*}{\Rightarrow} y$. The induction hypothesis applies to x, so $n_0(x) = n_1(x)$. The induction hypothesis also applies to y, so $n_0(y) = n_1(y)$. Hence, by construction, $n_0(w) = n_1(w)$ too. Thus $w \in L$.

We have shown that in all cases, if $w \in L(G)$ then $w \in L$. Thus the containment $L(G) \subseteq L$ is established.

Proof that $L \subseteq L(G)$: Let $w \in L$ be arbitrary. We must show that $w \in L(G)$. The proof is by induction on $|w|$.

- Base case ($|w| = 0$): Then $w = \varepsilon$. Then $w \in L(G)$ via the production $S \to \varepsilon$ in G.
- Inductive case ($|w| \geq 1$): The induction hypothesis is that for all words $x \in L$ with $|x| < |w|$, we have $x \in L(G)$.
- Note that all words in L have even length, since they have equal numbers of 0s and 1s.
- We will have four cases, depending on the two outside letters of w.

Four cases of the induction proof

1. $w = 0x1$.
    - Since $w \in L$, therefore $n_0(w) = n_1(w)$ by the definition of L.
    - But then $n_0(x) = n_1(x)$ by construction, and so $x \in L$.
    - Also by construction, $|x| < |w|$.
    - Thus by the induction hypothesis, $x \in L(G)$, i.e. $S \overset{*}{\Rightarrow} x$.
    - Then we can derive w in G via

$$S \Rightarrow 0S1 \overset{*}{\Rightarrow} 0x1 = w, \text{ so } w \in L(G).$$

2. $w = 1x0$. Same argument, instead using the rule $S \to 1S0$ to start.
3. $w = 0x0$. We will use the rule $S \to SS$ here to produce w (see below).

4. $w = 1x1$. Here, we will use the rule $S \to SS$ analogously (see below).

For cases 3 and 4, must show that w can be decomposed into two parts $w = yz$, both of which are in $L(G)$.

Finishing cases 3 and 4 Claim: If $w = 0x0$ and $n_0(w) = n_1(w)$, then we can write $w = yz$, where $n_0(y) = n_1(y)$, $y \neq \varepsilon$, and $z \neq \varepsilon$.

Why is this useful?

- Decompose w into two parts that are both shorter and in L.
- By the inductive hypothesis, they are then both also in $L(G)$.
- Use the $S \to SS$ rule to start the derivation.

Why is it true?

- Basic idea: look at the balance between the number of 1s and 0s in prefixes of w.
- The 1-letter prefix of w is 0, which has one more 0 than 1s.
- The $(|w| - 1)$-letter prefix of w is $0x$, which has one fewer 0 than 1s. (Why? We will wind up evening out at the last letter, which is 0.)
- This balance between 0s and 1s shifts by 1 each letter. It eventually goes from $+1$ to -1.
- So at some point strictly in between, the balance is zero.

End of cases 3 and 4 Definitions:

- Let p_i be the i-letter prefix of w.
- Let $b_i = n_0(p_i) - n_1(p_i)$.

With this in mind:

- $b_1 = 1$, since $p_1 = 0$.
- $b_{|w|-1} = -1$, since $p_{|w|-1} = 0x$, where $0x0 \in L$, and
- $b_{|w|} = 0$, since $p_{|w|} = 0x0 \in L$.
- For any i, $b_i = b_{i-1} \pm 1$, depending on whether the i^{th} character is 1 or 0.
- Since we must go from $+1$ to -1 by steps of 1, there must be some i satisfying $1 < i < |w| - 1$ such that $b_i = 0$.
- Decompose $w = yz$, then, taking y to be the i-letter prefix of w and z the rest of w.
- The two substrings y and z are both shorter (and both in L by construction), so we can use $S \to SS \overset{*}{\Rightarrow} Sz \overset{*}{\Rightarrow} yz = w$ as our derivation for w.

Thus, $w \in L(G)$. (This works for Case 4 also, swapping 0 and 1.) $\square$

We are done!

- We were given the language $L = \{w \in \{0,1\}^* \mid n_0(w) = n_1(w)\}$, and told to show it is context free.
- We gave a grammar G, and asserted that $L(G) = L$.
  - We showed that $L(G) \subseteq L$, by showing that any word derived in G had an equal number of 0's and 1's.
  - We showed that $L \subseteq L(G)$, by showing that any word with an equal number of 0's and 1's could be derived in G.
- Hence, L is context free.
- (You can easily prove that L is not regular, using the pumping lemma on $0^n 1^n$.)

9 Lecture 09

Outline

1. Parse Trees - M5 26-33
2. Ambiguity in Context-Free Grammars - Definitions - M5 34-43

9.1 Parse Trees

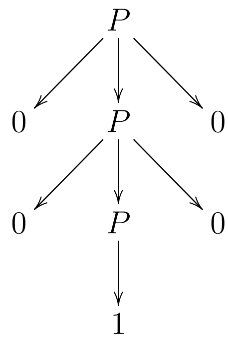
Key Idea: A **parse tree** is a visual presentation of a derivation.
Example: Consider this grammar (which generates palindromes) from the slides:

$$G : P \to \varepsilon \mid 0 \mid 1 \mid 0P0 \mid 1P1.$$

Exercise: Give a rigourous proof that $L(G)$ is actually the language of palindromes. (A **palindrome** is a word that reads the same backwards as forward.)
The derivation in this grammar

$$P \Rightarrow 0P0 \Rightarrow 00P00 \Rightarrow 00100$$

is represented by the parse tree



Parts of the Tree:

- **Root** of the tree (top of the picture): some variable
- **Internal nodes** of the tree: variables generated by some production
- **Leaves** of the tree: variables, terminals, or ε, generated by some production

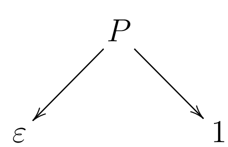
Definition 9.1.1. *Given a context-free grammar G, a rooted tree is a* ***parse tree*** *for a derivation in G if*

1. *the root is a variable in G,*
2. *the interior nodes are labelled by variables in G,*
3. *the leaves are either labelled by variables of G, terminals in the grammar G, or ε (If a leaf is labeled by ε, then it must be the only child of its parent), and*
4. *if there is an internal node labelled A, whose children are labelled $B_1, B_2, \ldots, B_k$ (for some $k \geq 1$), then there must be a production $A \to B_1 B_2 \cdots B_k$ in G.*

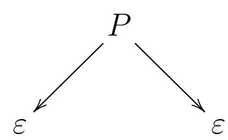
Any subtree of a parse tree is also a parse tree, unless it is a leaf.

Q & A

1. Why must a leaf labelled ε be the only child of its parent?
   A: If we had an ε-leaf, not an only child, the last rule in our definition of a parse tree requires that the production that made the ε-leaf have an extraneous ε in it, which would never be needed, e.g.



Then the grammar must have a production $P \to \varepsilon 1$. We agree to always simplify such rules: $P \to 1$. Similarly,



could just keep $P \to \varepsilon$. Because we want to describe our languages as succinctly as possible, we only use ε in a production like $S \to \varepsilon$. Hence the parse trees won't have ε-leaves as siblings of non-ε-leaves.

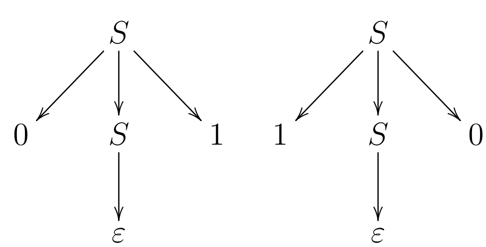
Definition 9.1.2. *Given a parse tree, the* ***yield*** *of the tree is the concatenation of the symbols at the leaves of the tree, in order from left to right. (N.B. Don't include ε in the concatenation.)*

Example: The yield of the above parse tree is 00100.

Definition 9.1.3. *A parse tree is* ***full*** *if its root is the start variable in the grammar, and if all its leaves are labelled with terminals or ε. In these trees, the yield is to a word in the language of the grammar.*

Remarks:

1. Order matters here. We read the yield of a parse tree along the leaves, from left to right. Recall this grammar from the proof of Theorem 8.1.6:
$$S \to \varepsilon \mid 0S1 \mid 1S0 \mid SS$$
These two parse trees over this grammar have different yields:



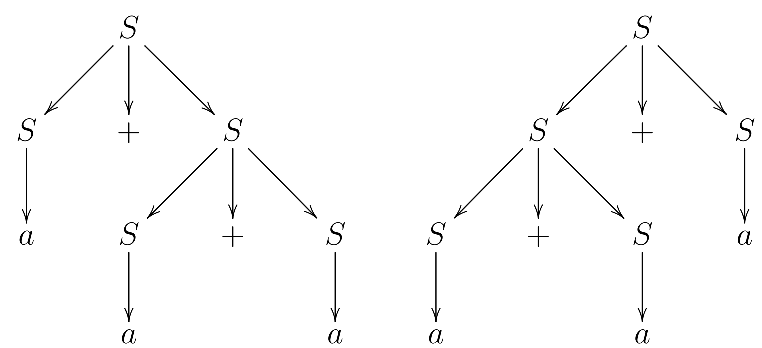
2. The above parse tree with yield 00100 is full.
3. We can go from derivations to parse trees and back.
4. As in the slides, we could formalize this correspondence, but we will not.

9.2 Ambiguity in Context-Free Grammars - Definitions

Definition 9.2.1. *A context-free grammar, G, is **ambiguous** if there is a word $w \in L(G)$ with more than one leftmost derivation (or equivalently more than one parse tree).*

Example: This grammar, given in the slides, is ambiguous:

$$G : S \to S * S \mid S + S \mid (S) \mid a$$



Both trees yield $a + a + a$, but the derivations are different!

Definition 9.2.2. *A context-free grammar G is **unambiguous** if it is not ambiguous (i.e. G is unambiguous if and only if every $x \in L(G)$ has a unique derivation in G).*

Definition 9.2.3. *A context-free language L is **inherently ambiguous** if every context-free grammar G such that $L(G) = L$ is ambiguous.*

Remarks:

1. It should be clear that for parsing purposes, unambiguous grammars are desirable.
2. If a CFL is inherently ambiguous, then there is no hope.
3. Sometimes the first grammar we might write down is ambiguous, but there is a non-ambiguous alternative grammar that generates the same language.
4. The above grammar is ambiguous, but its language is not inherently ambiguous. the example of replacing the grammar with a different, unambiguous grammar which generates the same language, demonstrates this.

Setup for the Example from the Slides:

1. We will show that the grammar $G : S \to S * S \mid S + S \mid (S) \mid a$ is ambiguous.
2. We will show that the new grammar G', with three variables:
   - E for expressions (which is the start variable): $E \to E + T \mid T$
   - T for terms, which can be added together to make expressions: $T \to T * F \mid F$
   - F for factors, which can be multiplied into products: either the single terminal a or a parenthesized expression: $F \to (E) \mid a$

   is unambiguous, and generates the same language.
3. Examples like this one tend to be tedious, but not difficult.
4. You will do an example of removing ambiguity like this one on A03.

Q & A

1. If G is ambiguous, can $L(G)$ be non-inherently ambiguous?
   A: Yes, per the current example. Remember, ambiguity applies to grammars; inherent ambiguity applies to languages.
2. Why is ambiguity bad?
   A:
   (a) CFGs / CFLs play a role in compilers. In compiling, ambiguity is bad.
   (b) In CS 360, we study formal languages (currently, CFLs). Suppose we want to prove that all words in some CFL have some property. It is good for us if every word in the CFL has a unique derivation. If multiple derivations are possible, then our proofs will get much more complicated.

10 Lecture 10

Outline

1. Ambiguity in Context-Free Grammars - Example - M5 44-67

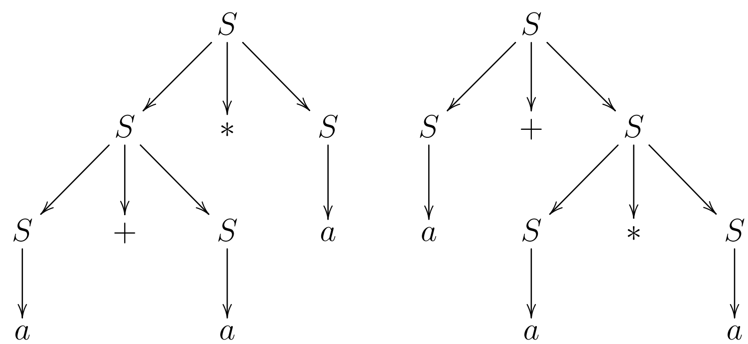
10.1 Ambiguity in Context-Free Grammars - Example

Consider the language of the grammar

$$G : S \to S * S \mid S + S \mid (S) \mid a$$

Where does this grammar acquire ambiguity? Two ways:

1. We can derive $a + a + a$ in two different ways: no sense of order in associativity. See the previous lecture for the corresponding diagram.
2. We can derive $a + a * a$ in two different ways: no sense of operator precedence.



Strategy to Remove Ambiguity:
Separate expressions being multiplied, added or parenthesized

- If we have the sum of two terms, we cannot use that as a factor in a product.
- The factors being multiplied in a product have to be either products themselves, or a or a parenthesized expression.
- If we have the sum of three terms, we have to sum the first two, and then sum this result with the third (e.g. $a+a+a$ means $(a+a)+a$).

A new grammar This idea suggests a grammar G' with three variables:

- E for **expressions** (which is the start variable): $E \to E + T \mid T$
- T for **terms**, which can be part of a sum: $T \to T * F \mid F$
- F for **factors**, which can be multiplied into products: either the single terminal a or a parenthesized expression: $F \to (E) \mid a$

(**Q:** Why not $E \to T + T \mid T$? **A**: Because then there is still no **unique** way to derive $a + a + a$. We need to make sure that there is **exactly** one way of generating every expression!)
This grammar includes the order of operations. It is also unambiguous and generates the same language.
Formal statement of our result

Theorem 10.1.1. *If G is the grammar*

$$S \to S + S \mid S * S \mid (S) \mid a,$$

and G' is the grammar with rules:

- $E \to E + T \mid T$,
- $T \to T * F \mid F$,
- $F \to (E) \mid a$,

then

1. G' *is unambiguous, and*
2. $L(G') = L(G)$.

Remark: This is **not** going to be a short proof.

Lemma 10.1.2. *First, consider* G'*:*

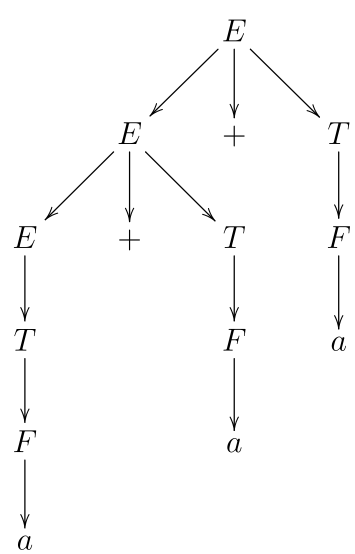
1. *If* $T \overset{*}{\Rightarrow} x$*, then* x *has no* $+$ *character outside of parentheses.*
2. *If* $F \overset{*}{\Rightarrow} x$*, then* x *has no* $+$ *or* $*$ *characters outside parentheses.*
3. *If* x *has no* $+$ *or* $*$ *characters outside parentheses, then* $F \overset{*}{\Rightarrow} x$.
4. *If* x *has no* $+$ *characters outside parentheses, then* $T \overset{*}{\Rightarrow} x$ *or* $F \overset{*}{\Rightarrow} x$.

Proof: This is clear from the rules of G'. Prove by structural induction on G' if you are not already convinced. $\square$

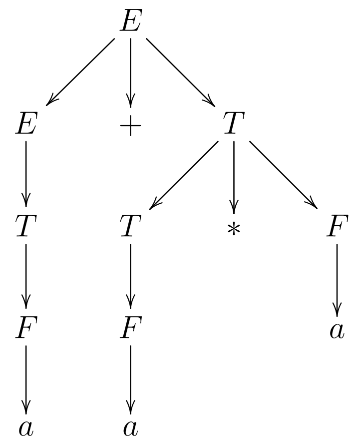
We will use these facts quite a lot.

Examples which are ambiguous in G, now unambiguous in G':

1. In this new grammar, $a + a + a$ has only one parse tree:



2. and similarly, $a + a * a$ has only one parse tree:



First: G' **is unambiguous** We will prove something stronger:

Lemma 10.1.3. *If, for some string x containing only alphabet symbols, we have $E \overset{*}{\Rightarrow} x$ or $T \overset{*}{\Rightarrow} x$ or $F \overset{*}{\Rightarrow} x$, then there is a unique left-most derivation for that string from E, T or F, respectively.*

Q & A

1. Are all of E, F and T starting variables?
   A: No. E is the starting variable.
2. Given the above answer, why do we care about F and T?
   A: Including all variables for the Lemma makes our induction hypothesis stronger. This turns out to be useful.

Proof. by induction on $|x|$:

- Base case ($|x| = 1$): Then $x = a$.
  - If $E \overset{*}{\Rightarrow} a$, then the only possible derivation is: $E \Rightarrow T \Rightarrow F \Rightarrow a$.
  - If $T \overset{*}{\Rightarrow} a$, then the only possible derivation is: $T \Rightarrow F \Rightarrow a$.
  - If $F \overset{*}{\Rightarrow} a$, then the only possible derivation is: $F \Rightarrow a$.
- Inductive case ($|x| > 1$): Inductive Hypothesis: Any y with $E \underset{G'}{\overset{*}{\Rightarrow}} y, T \underset{G'}{\overset{*}{\Rightarrow}} y$ or $F \underset{G'}{\overset{*}{\Rightarrow}} y$ and $|y| < |x|$ has a **unique** leftmost derivation from E, T or F, respectively.
- Now, let's look at $x \in L(G')$ and figure out how we got to where we are.

Three cases for the induction We will examine x and see what it has outside of parentheses:

1. x has a $+$ symbol outside of parentheses,
2. x has no $+$ symbol outside of parentheses in x, but has a $*$ outside of parentheses, or
3. x has neither a $+$ symbol nor a $*$ symbol outside of parentheses.

Case 1 Assume x has a $+$ symbol outside of parentheses.
Basic idea: Writing $x = y + z$, argue that we must start with the rule $E \to E + T$ where $E \stackrel{*}{\Rightarrow} y$ and $T \stackrel{*}{\Rightarrow} z$, and then argue that y and z must have unique derivations, by Lemma 10.1.2.

- If x has a $+$ outside of parentheses, then $T \stackrel{*}{\not\Rightarrow} x$ and $F \stackrel{*}{\not\Rightarrow} x$ (Lemma 10.1.2 for both).
- The only remaining possibility is that $E \stackrel{*}{\Rightarrow} x$.
- So the first rule used in any derivation of x in G' must be $E \to E + T$.
- We know that $x \in L(G')$, so there must be some way, now, to generate $x = y + z$, where $E \stackrel{*}{\Rightarrow} y$ and $T \stackrel{*}{\Rightarrow} z$.
- But by our induction hypothesis, the ways of generating y and z are unique, so for this decomposition $x = y + z$, there exists exactly one way of generating x.
- **Q:** Could we choose y and z in different ways?
- **A:** No: we know that y must finish immediately before the last $+$ symbol outside parentheses in x.
  - If not, then z has a $+$ symbol outside parentheses, and thus by Lemma 10.1.2 z cannot be generated from T.
  - But we know that $T \stackrel{*}{\Rightarrow} z$.
- So, for this case, the only leftmost derivation for x is: $E \Rightarrow E + T \stackrel{*}{\Rightarrow} y + T \stackrel{*}{\Rightarrow} y + z$, and x is unambiguously derived.

Case 2 Assume x has no $+$ outside parentheses, but does have a $*$ outside parentheses.

- We cannot start with the $E \to E + T$ rule, so we must start with $E \to T$ instead.
- But since there is a $*$ outside parentheses, by Lemma 10.1.2, $F \stackrel{*}{\not\Rightarrow} x$.
- So we cannot use $T \to F$ and we must use the $T \to T * F$ rule instead. We are going to thus be decomposing x into $x = y * z$.
- As in the proof for Case 1 (since $F \stackrel{*}{\Rightarrow} z$), z must have no $*$ outside parentheses, so it is uniquely chosen.
- Again, for $x = y * z$, y and z have unique derivations by induction, and

hence there is a unique derivation for x.

Case 3 Assume x has neither * nor + outside parentheses.

- But then $x = (y)$ for some y, since we are **not** in the base case, namely the case where $x = a$.
- We must start the derivation with $E \Rightarrow T \Rightarrow F \Rightarrow (E)$.
- Then we follow the derivation for y, which is unique by the induction hypothesis.

This handles all three possibilities for a word in $L(G')$. In all cases, they have a unique derivation.
Hence, the grammar G' is unambiguous. □

That is the proof that the new grammar is unambiguous. This proves part 1 of Theorem 10.1.1.
What about the proof that $L(G) = L(G')$ (part 2 of Theorem 10.1.1)?

Proof. Proof that $L(G') \subseteq L(G)$: Let $x \in L(G')$ be arbitrary. The proof is by induction on $|x|$.
Base case ($|x| = 1$): If , then $x = a$, since that is the only 1-letter word in $L(G')$, and it is the only 1-letter word in $L(G)$, too.
Inductive case ($|x| > 1$): Inductive Hypothesis: every $y \in L(G')$ such that $|y| < |x|$ satisfies $y \in L(G)$. Consider the **unique** derivation of x in G'.
How is x derived? Three cases for the **start** of the unique derivation of x in G':

1. $E \to E + T \stackrel{*}{\Rightarrow} x$.
2. $E \Rightarrow T \Rightarrow T * F \Rightarrow \stackrel{*}{\Rightarrow} x$
3. $E \Rightarrow T \Rightarrow F \Rightarrow (E) \stackrel{*}{\Rightarrow} x$

Case 1: x derives from $E \to E + T$

- Write $x = y + z$, where $E \stackrel{*}{\Rightarrow} y$ and $T \stackrel{*}{\Rightarrow} z$.
- Since $E \stackrel{*}{\Rightarrow} y$, therefore $y \in L(G')$.
- Since $E \Rightarrow T \stackrel{*}{\Rightarrow} y$, therefore $z \in L(G')$.
- We can derive z in G' using $E \Rightarrow T \stackrel{*}{\Rightarrow} z$.
- Both y and z are in $L(G')$ and are strictly shorter than x, so they are both in $L(G)$ by our inductive hypothesis.
- Derive x in $L(G)$ via

$$S \Rightarrow S + S \stackrel{*}{\Rightarrow} y + S \stackrel{*}{\Rightarrow} y + z = x,$$

so that $x \in L(G)$.

Case 2: x derives from $E \Rightarrow T \Rightarrow T * F \Rightarrow \cdots$

- Then $x = y * z$, where $T \stackrel{*}{\Rightarrow} y$ and $F \stackrel{*}{\Rightarrow} z$.
- Since $T \stackrel{*}{\Rightarrow} y$, we can derive y via $E \Rightarrow T \stackrel{*}{\Rightarrow} y$, and therefore $E \stackrel{*}{\Rightarrow} y$.
- Since $F \stackrel{*}{\Rightarrow} z$, we can derive z via $E \Rightarrow T \Rightarrow F \stackrel{*}{\Rightarrow} z$, and therefore $E \stackrel{*}{\Rightarrow} z$.
- So y and z are in $L(G')$ and are shorter than x.
- By the inductive hypothesis, y and z are in $L(G)$, and we can derive x in G by
$$S \Rightarrow S * S \stackrel{*}{\Rightarrow} y * S \stackrel{*}{\Rightarrow} y * z = x.$$

Case 3: x derives from $E \Rightarrow T \Rightarrow F \Rightarrow (E) \Rightarrow \cdots$
- Then $x = (y)$, where $E \stackrel{*}{\Rightarrow} y$.
- The induction hypothesis applied to y says $S \stackrel{*}{\Rightarrow} y$.
- Therefore we can derive x in G via

$$S \Rightarrow (S) \stackrel{*}{\Rightarrow} (y) = x.$$

So we see that $L(G') \subseteq L(G)$.
Proof that $L(G') \supseteq L(G)$: Let $x \in L(G)$ be arbitrary. The proof is by induction on $|x|$.
Base case ($|x| = 1$): Then $x = a \in L(G')$; seen before.
Inductive case ($|x| > 1$):
- Inductive Hypothesis: Assume all words $y \in L(G)$ such that $|y| < |x|$ satisfy $y \in L(G')$.
- Then $x = y + z$ or $x = y * z$ or $x = (y)$ for y and z also in $L(G)$.
- We will have three cases, depending on how x can be derived in G.
- As derivations in G need not be unique, we need to carefully state what the cases are.

Cases for the Inductive part of the proof:
1. x has at least one derivation of the form $S \Rightarrow (S) \stackrel{*}{\Rightarrow} (y)$, where $x = (y)$.
2. x has no such derivation, but has at least one derivation of the form $S \Rightarrow S + S \stackrel{*}{\Rightarrow} y + z$, where $x = y + z$.
3. The only derivations for x have the form $S \rightarrow S * S \stackrel{*}{\Rightarrow} x$.

Case 1: x has at least one derivation of the form $S \Rightarrow (S) \stackrel{*}{\Rightarrow} (y)$, where $x = (y)$: Suppose that one derivation for x in G is:

$$S \Rightarrow (S) \stackrel{*}{\Rightarrow} (y).$$

- Then in G' we can use the derivation: $E \Rightarrow T \Rightarrow F \Rightarrow (E) \cdots$

- Since $S \overset{*}{\underset{G}{\Rightarrow}} y$, and $|y| < |x|$, our inductive hypothesis tells us that $y \in L(G')$, i.e. $E \overset{*}{\underset{G'}{\Rightarrow}} y$.
- Therefore in G' we have $E \Rightarrow T \Rightarrow F \Rightarrow (E) \overset{*}{\Rightarrow} (y) = x$, which witnesses the fact that $x \in L(G')$.

Case 2: $S \Rightarrow S + S \overset{*}{\Rightarrow} y + z$**, where** $x = y + z$**:** Suppose that x has no derivation as in Case 1, but has at least one derivation in G of the form $S \Rightarrow S + S \overset{*}{\Rightarrow} y + z$, where $x = y + z$.

- **Problem**: we want to start the derivation of x in G' using the production: $E \to E + T$.
- By the induction hypothesis, $E \overset{*}{\underset{G'}{\Rightarrow}} y$ and $E \overset{*}{\underset{G'}{\Rightarrow}} z$, but to apply the desired production in G', we need to know that $T \overset{*}{\underset{G'}{\Rightarrow}} z$.
- **Idea**: Choose y as long as possible, so that z has no $+$ symbols outside of parentheses.
- Then, as above, we still have $E \overset{*}{\Rightarrow} y$, but now we can only use $E \Rightarrow T \overset{*}{\Rightarrow} z$ to derive z in G'.
- But then we do have $T \overset{*}{\underset{G'}{\Rightarrow}} z$, as required.
- Putting it all together, in G' we have $E \Rightarrow E + T \overset{*}{\Rightarrow} y + z = x$.
- This witnesses the fact that x is in $L(G')$.

Case 3: $S \Rightarrow S * S$**, where** $x = y * z$**:** This last case is quite similar.

- Suppose that the only derivations for x start by using the production $S \to S * S$.
- Because x has no derivation beginning with $S \to S + S$, therefore every $+$ in x must be inside parentheses.
- If x contains an exposed $+$, then we can write $x = y + z$, with $S \overset{*}{\Rightarrow} y$ and $S \overset{*}{\Rightarrow} z$, so that we can always find a derivation for x which starts with $S \to S + S$, contradicting the case that we are in.
- Write $x = y * z$, choosing y as long as possible such that we can still derive both y and z from S in G.
- Then y is a word in $L(G)$, shorter than x, and hence by the induction hypothesis is in $L(G')$.
- But then y must be derivable in G' via $E \Rightarrow T \overset{*}{\Rightarrow} y$, since y has no exposed $+$ characters.
- Also, z must have no exposed $+$ or $*$ characters, yet still be in $L(G)$.
- Therefore z is derivable in G' by $E \Rightarrow T \Rightarrow F \overset{*}{\Rightarrow} z$.
- So finally in G' we can derive x by $E \Rightarrow T \Rightarrow T * F \overset{*}{\Rightarrow} y * F \overset{*}{\Rightarrow} y * z$.

- Thus $x \in L(G')$.

$\square$

Recap of the proof: Overall, we have shown:

1. G' is unambiguous
2. Words in $L(G')$ are in $L(G)$.
3. Words in $L(G)$ are in $L(G')$.

(All 3 proofs by induction.)
The third proof was probably hardest: the content was to reconstruct the way in G' to derive each word of $L(G)$.
Hence, $L(G)$ is **not** inherently ambiguous: G' is an unambiguous grammar for it.
What does this show us? Ambiguity can, sometimes, be removed:

- Identify the source of ambiguity.
- Re-organize to identify precedence or other desired grammar rules.
- Such proofs are long, but useful: we really do need unambiguous grammars in practice.

11 Lecture 11

Outline

1. Introduction to Pushdown Automata - M6 1-10
2. Computations in a PDA - M6 11-16
3. Bad things in PDAs - M6 17-18
4. Language of a PDA - M6 19-25

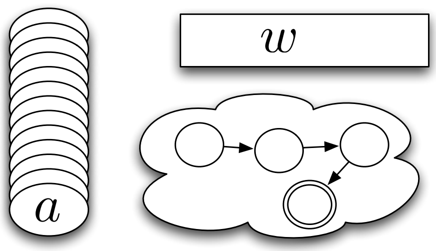
11.1 Introduction to Pushdown Automata

From an ε-NFA to a PDA
Add memory to an ε-NFA, in the form of a **stack**. Each transition is of the form:

- If
  - top letter on the memory stack is b, and
  - the next letter in the input word is a (or ε, if we take an ε-transition), and
  - the controlling FA is in state q,
- then:

– Go to state r,
– Pop the b off the top of the memory stack,
– Consume the letter a from the input (again, a might be ε), and
– Push a **word** w onto the top of the memory stack.

- We will define the **stack alphabet** as part of the definition of the PDA.



Remarks:

1. This is modelled on an ε-NFA, which is **inherently nondeterministic**. Therefore every thread running in a PDA has its own stack, independent of the stacks in the other threads.

More details about how PDA work

The stack must never become empty:

- There is a special "stack empty" symbol Z_0.
  – If Z_0 is on the top of the stack, we make sure to re-push at least Z_0 onto the stack as we take a transition.
- The string put on the stack
  – may be empty (then the stack becomes 1 symbol shorter), or long,
  – does not have to just be 0 or 1 letters long!
  – *must* be of finite length.

(We will see later that deterministic PDAs are **less powerful** than nondeterministic ones!)

Definition 11.1.1. *A* ***pushdown automaton (PDA)*** *is a 7-tuple,* $M = (\Sigma, Q, F, q_0, \Gamma, Z_0, \delta)$, *where:*

- Σ *is an alphabet*
- Q *is a finite set of states for FA control*
- F *is a set of accept states for FA control*
- q_0 *is the start state for FA control*
- Γ *is a finite* ***stack alphabet*** *(often it is just* $\Gamma = \{Z_0\} \cup \Sigma$, *but* Γ *can be any finite set)*
- Z_0 *is the special empty stack symbol (the stack is initialized with just*

Z_0 on it)

- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to$ *finite subsets of* $Q \times \Gamma^*$ *is the transition function*

Remarks on Definition 11.1.1:

1. $Z_0 \notin \Sigma$.
2. **Q:** Why does δ map into finite subsets of $\{Q \times \Gamma^*\}$? **A:** There are an infinite number of words in Γ^* that we could put on the stack! The machine description must be finite.

PDA Diagrams

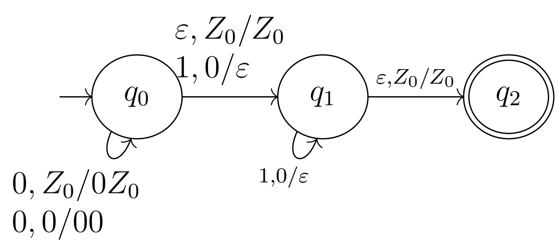
We draw PDAs like DFAs, with more robust labels of edges:

1. Edges are labelled by both the input letter (e.g. a or ε) **and** the top symbol of the stack (e.g. b); these are separated by a comma.
2. We also must show what word w is pushed onto the stack: this follows a "/" character. The beginning of the string is the *top* of the stack after the transition occurs. Examples:
   (a) if the stack is empty, and we push $w = 10$ onto the stack, then 1 is on top of the stack and 0 is below 1.
   (b) If we read Z_0 from the stack, then push $01Z_0$ onto the stack, then we imagine the stack contents as

0
1
Z_0

3. It is possible that ε is pushed onto the stack: this makes the stack become shorter.

Example:



Remarks:

1. Naming this PDA P, we will see later that $L(P) = \{0^i1^i \mid i \geq 0\}$.
2. By the end of M6, we will prove the analog of Kleene's Theorem for CFLs and PDAs. By Kleene's Theorem, $L(P)$ is a CFL.
3. <u>Exercise:</u> Give a CFG, G, such that $L(G) = L(P)$.

11.2 Computations in a PDA

Definition 11.2.1. *Given a PDA, its* ***instantaneous description*** *is a triple* (q, w, γ)*, where*

1. *q is the state the machine is in,*
2. *w is the word remaining on input, and*
3. *γ is the stack contents.*

Remarks:

1. PDAs are non-deterministic: they can have multiple threads.
2. Definition 11.2.1 describes one thread only.
3. We don't attempt to define $\hat{\hat{\delta}}$ for PDAs. Because of the stack, it is too cumbersome. We have to write down each thread separately.

We need a way to characterize one step in the PDA's computation:

- This is more complicated than the $\hat{\hat{\delta}}$ notation for finite automata: we need to show what happens to the stack.
- Notation describes what could follow the current configuration of the computation.
- Our new notation describes "one path" of the PDA's computation: to simulate the whole PDA, you would need to present all nondeterministic possibilities.

Transitions in the PDA

What happens if we go forward one step?

- Let (q, w, γ) and (p, z, ζ) be two instantaneous descriptions of a PDA's configuration.
- From the first to the second in one transition: $(q, w, \gamma) \vdash (p, z, \zeta)$.
- What does that really mean? Two possibilities:
    1. If $\gamma = X\beta$ and $w = az$, where
        (a) $|X| = 1$, and
        (b) $|a| = 1$,
        (c) then $\delta(q, a, X)$ contains (p, α), where $\zeta = \alpha\beta$.
        (d) I.e. there is a transition:

$$\textcircled{q} \xrightarrow{a,X/\alpha} \textcircled{p}$$

        Remember also that α is of finite length.
    2. Or if $\gamma = X\beta$ and $w = \varepsilon z = z$, where
        (a) $|X| = 1$,

   (b) then $\delta(q, \varepsilon, X)$ contains (p, α), where $\zeta = \alpha\beta$.
   (c) I.e. there is a transition

$$q \xrightarrow{\varepsilon,X/\alpha} p$$

3. Then we write $(q, w, \gamma) \vdash (p, z, \zeta)$.
4. Read $\vdash$ as "produces in one step".

If the machine P needs to be indicated explicitly, then write $I \underset{P}{\vdash} J$.

Longer derivations

1 step in P: $(q, x, \beta) \underset{P}{\vdash} (q', x', \alpha)$.

An arbitrary (finite) number of steps: $(q, x, \beta) \overset{*}{\underset{P}{\vdash}} (q', x', \alpha)$.

Definition 11.2.2. *Proper inductive definition of* $I \overset{*}{\underset{P}{\vdash}} J$*:*

1. *Base case: For any instantaneous description* I*,* $I \overset{*}{\underset{P}{\vdash}} I$.
2. *Inductive case: If* $I \underset{P}{\vdash} K$ *and* $K \overset{*}{\underset{P}{\vdash}} J$*, then* $I \overset{*}{\underset{P}{\vdash}} J$.

(I.e. there exists a finite sequence of instantaneous descriptions $K_1, K_2, \ldots, K_n$ *such that* $I = K_1$*,* $J = K_n$ *and for all* $i = 1, \ldots, n-1$*,* $K_i \underset{P}{\vdash} K_{i+1}$*.)*

Reminders:

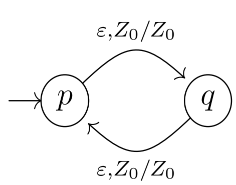
1. PDAs are non-deterministic. They can have multiple threads.
2. This describes <u>one thread only</u>.

We will **not** attempt to define an analogue to $\hat{\delta}$ for PDAs.

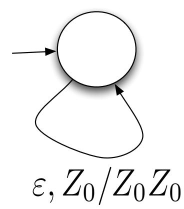
11.3 Bad things in PDAs

There are two bad kinds of events too worry about here. Both are also possible with ε-NFAs.

1. **Running forever, using ε-transitions:**
   - There are only a finite number of possible states we can reach following ε-transitions, and they will cycle. E.g.



- But now we also can grow the stack, so the configurations are



different at each step.

- $(q_0, x, Z_0) \vdash (q_0, x, Z_0Z_0) \vdash (q_0, x, Z_0Z_0Z_0) \vdash (q_0, x, Z_0Z_0Z_0Z_0) \overset{*}{\vdash} \ldots$
- One thread can run forever, or the machine can spawn infinitely many threads. This example produces <u>infinitely many instantaneous descriptions</u>, because the stack keeps growing.

Remark: This is a DPDA, so even DPDAs can run forever.

2. **Crashing** A PDA thread **crashes** if there are no available transitions from the current instantaneous description, or if the stack is empty with some input characters remaining.
   - <u>No available transitions:</u> if the current instantaneous description is (q, x, z), where $x = aw$ and $z = X\beta$ for some alphabet symbol a and stack symbol X, and if the sets $\delta(q, a, X)$ and $\delta(q, \varepsilon, X)$ are both empty, then the PDA can make no transitions and therefore crashes.
   - <u>Empty stack:</u> If the current instantaneous description is (q, x, ε) for some $x \neq \varepsilon$, then the PDA can make no transitions and therefore crashes.
     - It is supposed to remove the top stack symbol as it makes its next transition, but there is no top stack symbol!
     - Note that, if the machine arrives at an instantaneous description $(q, \varepsilon, \varepsilon)$, then the machine will not crash; it will
       - accept if $q \in F$, and
       - reject otherwise.

11.4 Language of a PDA

Useful Facts:

A valid PDA computation remains valid if we:

1. Append string $w \in \Sigma^*$ to the end of the input for all K_i.
2. Append string $\gamma \in \Gamma^*$ to the end of the stack for all K_i.

3. Remove an unused suffix from the input for all K_i.

Why do we care?

- Let us isolate part of a computation.
- Adding to the end of the stack will help us with some theorems about the equivalence of PDAs and CFGs.

We can prove the first two principles in the same theorem:

Theorem 11.4.1. *Suppose that for a given PDA P, $(q, x, \alpha) \overset{*}{\underset{P}{\vdash}} (p, y, \beta)$. Then for any strings $w \in \Sigma^*$ and $\gamma \in \Gamma^*$, it is also the case that*

1. $(q, xw, \alpha) \overset{*}{\underset{P}{\vdash}} (p, yw, \beta)$, *and*
2. $(q, x, \alpha\gamma) \overset{*}{\underset{P}{\vdash}} (p, y, \beta\gamma)$.

Proof. Consider all transitions that show: $(q, x, \alpha) \overset{*}{\underset{P}{\vdash}} (p, y, \beta)$.

1. None of these transitions uses characters of w or γ.
2. As such, each transition is valid in the modified computation.

□

Removing common suffixes Similarly, if we remove a common suffix from the input, the computation remains valid:

Theorem 11.4.2. *If $(q, xw, \alpha) \overset{*}{\underset{P}{\vdash}} (p, yw, \beta)$, then $(q, x, \alpha) \overset{*}{\underset{P}{\vdash}} (p, y, \beta)$.*

Proof. This is easily shown by following all of the transitions in the first derivation.

1. None of them consumes any of the letters of w, since we can only remove symbols from the input of a PDA, never add to the input.
2. As such, each transition is also valid in the modified computation.

□

How does a PDA accept?

Definition 11.4.3. *A PDA, P **accepts** word w if there exists a valid computation*

$$(q_0, w, Z_0) \overset{*}{\underset{P}{\vdash}} (q, \varepsilon, y),$$

for some state $q \in F$, and some $y \in \Gamma^*$.

I.e. at least one thread in P reaches a final state by the end of input. It does not matter if other threads reject or crash; provided one thread accepts, P accepts w.

- This is called **acceptance by final state**.
- Soon we will accept if the stack empties at the same moment that the last input symbol has been processed (called **acceptance by empty stack**).

Definition 11.4.4. *The **language of a PDA** P is*

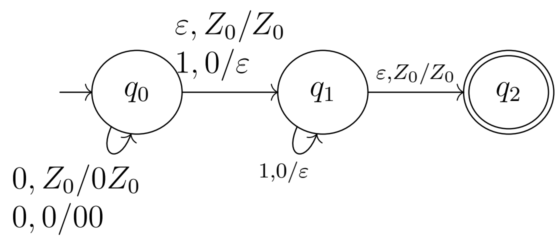
$$L(P) = \{w \in \Sigma^* \mid P \text{ accepts } w\},$$

as in Definition 11.4.3.

An example: $L = \{0^i 1^i\}$

A PDA for $L = \{0^i 1^i \mid i \geq 0\}$:

- Recall that this language is not regular.
- PDAs can be hard to draw!
- This one has three states:
  - q_0 for pushing 0s onto the stack, while reading them from the input word,
  - q_1 for popping 0s off of the stack, while reading corresponding 1s from the input word, and
  - q_2 for accepting at end of word.



How does this work?

For words in L:

- Start with stack Z_0.
- Add 0 symbols from w to the stack.
- ... until we reach the first 1 symbol. Then, we absorb 0s from the stack and corresponding 1s the input.
- ... until we are done, and then we follow the ε-transition to q_2 and accept.

For example, on the input 0011, a computation witnessing acceptance is:

$$\begin{aligned}(q_0, 0011, Z_0) &\vdash (q_0, 011, 0Z_0)\\ &\vdash (q_0, 11, 00Z_0)\\ &\vdash (q_1, 1, 0Z_0)\\ &\vdash (q_1, \varepsilon, Z_0)\\ &\vdash (q_2, \varepsilon, Z_0)\end{aligned}$$

Remark: Any stack contents are fine for acceptance by final state.
Some formality

1. Only words $w \in 0^*1^*$ can have the property that $(q_0, w, y) \overset{*}{\vdash} (q_2, \varepsilon, z)$ for any strings y and z, since we transition from q_0 to q_1 by consuming a word from $0^*(1+\varepsilon)$, and we transition from q_1 to q_2 by consuming a word from $1^*\varepsilon$; the concatenation of these is 0^*1^*.
2. The only valid computation after reading in 0^i for $i > 0$ is $(q_0, 0^iw, Z_0) \overset{*}{\vdash} (q_0, w, 0^iZ_0)$.
3. Now consider what occurs upon processing 0^i1^k.
4. If $k < i$, then all threads crash after reaching q_1. So from now on, we will consider $k \geq i$.
5. The only valid computation after reading in 0^i1^k is $(q_0, 0^i1^kw, Z_0) \overset{*}{\vdash} (q_1, w, 0^{i-k}Z_0)$, unless $k = i$, at which point we can transition to (q_2, ε, Z_0). If $k > i$, then the machine has no valid computations that read the first $i + k$ symbols.
6. We can only accept if after reading in 0^i1^i, there are no more symbols to read, so we need not fear accepting 0^i1^k when $k > i$.

Another example: a PDA for palindromes See the slides for the details.
Moral: By the Theorem at the end of M6, the language of palindromes is a CFL.

12 Lecture 12

Outline

1. Acceptance By Empty Stack - M6 31-36

12.1 Acceptance By Empty Stack

Motivation: By the end of Module 6, we will establish a correspondence between CFLs and PDA languages. It will turn out that establishing this correspondence will be easier with acceptance via empty stack than via final state.

Current model of acceptance:

- The language of the machine is

$$L(P) = \left\{ w \in \Sigma^* \mid \exists \text{ a computation } (q_0, w, Z_0) \overset{*}{\underset{P}{\vdash}} (p, \varepsilon, \alpha) \text{ for some } p \in F, \alpha \in \Gamma^* \right\}$$

New model of acceptance, by **empty stack**:

- The language of the machine is

$$N(P) = \left\{ w \in \Sigma^* \mid \exists \text{ a computation } (q_0, w, Z_0) \overset{*}{\underset{P}{\vdash}} (p, \varepsilon, \varepsilon) \text{ for } \textbf{any} \text{ state } p \right\}.$$

This is called **acceptance by empty stack**, for obvious reasons.

A PDA accepting by empty stack does not have accept states, so is a 6-tuple: $(\Sigma, Q, q_0, \Gamma, Z_0, \delta)$.

Key fact about acceptance by empty stack

Theorem 12.1.1. *1. Suppose that P_F is a PDA that accepts by final state. Then there exists a PDA, P_N, which accepts $L(P_F)$ by empty stack.*

2. Suppose that P_N is a PDA that accepts by empty stack. Then there exists a PDA, P_F, which accepts $N(P_N)$ by final state.

Remark on Why We Care About Theorem 12.1.1 We will use acceptance by empty stack to show that PDAs accept exactly the class of context-free languages.

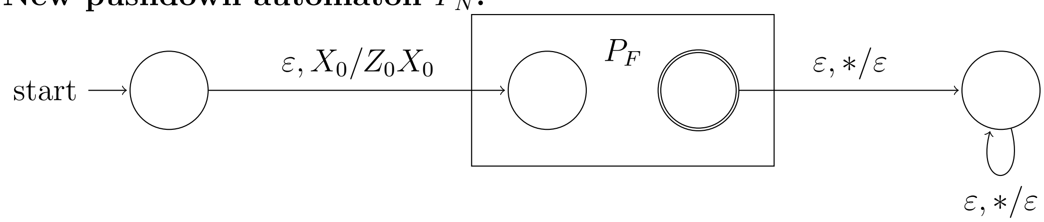
Proof. **From final state to empty stack**

Suppose we are given a PDA P_F whose language, using acceptance by final state, is L. We will construct a new PDA, P_N, accepting by empty stack, such that P_N accepts exactly L.

- To make P_N accept whenever P_F accepts:
  - From every accept state in P_F, take an ε-transition to a "drain" state that empties the stack. The drain state has no outgoing transitions.

- To prevent P_N from accepting whenever P_F does not accept:
  - If there are input letters remaining when the above ε-transition is taken, then the thread crashes.
  - To handle P_F's stack correctly, create a new empty stack symbol, X_0 in P_N. Push X_0 onto P_N's stack before pushing Z_0 to start simulating P_F.
  - It is crucial that X_0 **not** be in the stack alphabet for P_F.
  - We only pop the X_0 character when we are transitioning to, or already in, our new "drain" state.
  - I.e. we need X_0 so that, if P_F crashes because of empty stack, the constructed machine P_N will not accept.
  - All computations in P_F happen as before, so we can only remove Z_0, and then X_0, from the stack when P_F would have already accepted.

New pushdown automaton P_N:



Remark: Both $*$s here include X_0 in the set of all stack symbols in the constructed machine, P_N.

From empty stack to final state

A similar technique can be used to construct a PDA, P_F, with $L(P_F) = L$, given a PDA P_N with $N(P_N) = L$.

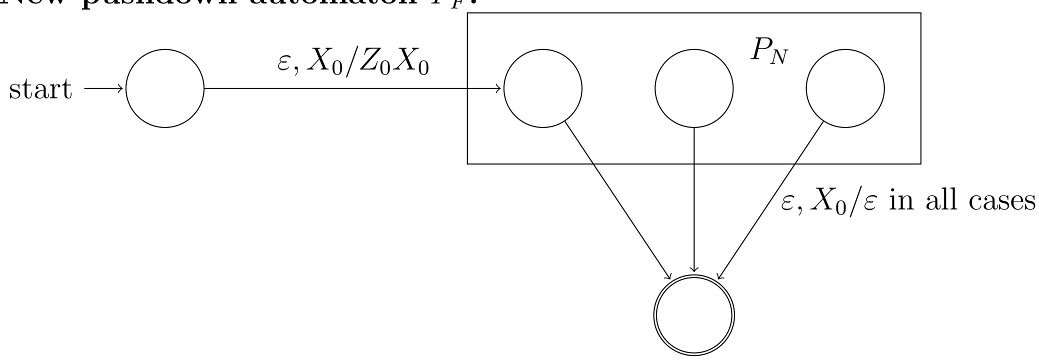
Again, we need a special character X_0 added to the end of the stack that only gets removed after P_N would have accepted.

- Again, it is crucial that X_0 **not** be in the stack alphabet for P_N.
- Here, the X_0 character alerts us that we have removed all of the stack characters inside of P_N (so we should accept, provided the input is also used up).
- Without this special symbol, we would crash trying to detect empty stack.
- When we see this special character, X_0 in **any** state, we take an ε-transition to the machine's only final state. Again, the final state has no outgoing transitions.

To ensure that the new PDA P_F only accepts (by final state) the words P_N accepts:

- P_F accepts precisely when we exhaust the input string at the same moment that we empty the stack.
- If we have not processed the entire input word, the machine crashes instead of accepting.

New pushdown automaton P_F:



Note: We have not rigourously proved either construction. Both proofs are in the text, not especially enlightening. □

13 Lecture 13

Outline

1. Equivalence of PDAs and CFGs - M6 37-56

13.1 Equivalence of PDAs and CFLs

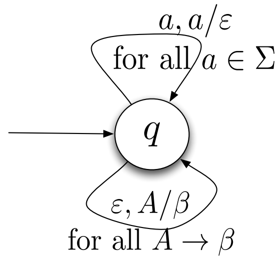
Theorem 13.1.1. *1. Given a CFG G whose language is L, there exists a PDA P such that $N(P) = L$.*
2. Given a PDA P where $N(P) = L$, there exists a CFG G such that $L(G) = L$.

Note: We will prove using acceptance by empty stack in both cases.
How to construct a PDA from a grammar
Start with the grammar $G = (V, T, P, S)$. Construct the PDA

$$P = \{T, \{q\}, q, V \cup T, S, \delta\},$$

where δ has the following members:

- For all rules $A \to \beta$ in the grammar, $\delta(q, \varepsilon, A)$ includes (q, β). There are no other members of $\delta(q, \varepsilon, A)$.
- For all terminals $a \in T$, $\delta(q, a, a) = \{(q, \varepsilon)\}$.
- All other values of $\delta(q, x, y)$ are $\emptyset$.



Theorem: This PDA, P, accepts $L(G)$ by empty stack. Two parts: $L(G) \subseteq N(P)$, and $N(P) \subseteq L(G)$. We will begin with showing that $L(G) \subseteq N(P)$.

- Let $w \in L(G)$ be arbitrary.
- Then there exists some leftmost derivation for w in G :

$$S = \gamma_1 \underset{lm}{\Rightarrow} \gamma_2 \underset{lm}{\Rightarrow} \gamma_3 \underset{lm}{\Rightarrow} \cdots \underset{lm}{\Rightarrow} \gamma_n = w.$$

- For all $1 \le i < n$, γ_i has a a leftmost variable.
  - For $1 \le i < n$, write $\gamma_i = x_i \alpha_i$, where the x_i prefix is all terminals and α_i begins with the leftmost variable, A_i, in γ_i. By construction, each x_i is a prefix of w.
  - Write $w = x_i y_i$, for some string $y_i \in \Sigma^*$.
- **I Claim that:** In the PDA, because the transitions are defined from the productions of G, we have $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, y_i, \alpha_i)$ for all $1 \le i \le n$. (We prove this claim rigourously, below.)
- Then in particular, taking $i = n$ gives $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, \varepsilon, \varepsilon)$, and so $w \in N(P)$, because P accepts w by empty stack.

<u>Proof of the claim</u>, by induction on i:

- Base case ($i = 1$): $x_1 = \varepsilon, y_1 = w, \alpha_1 = S$, so we have that $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, y_i, \alpha_i)$ holds trivially.
- Inductive case ($1 < i \le n$): The inductive hypothesis is that $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, y_j, \alpha_j)$, for all $j < i$.
  - By the induction hypothesis $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, y_{i-1}, \alpha_{i-1})$, where
    * $w = x_{i-1} y_{i-1}$,

    * $\gamma_{i-1} = x_{i-1}\alpha_{i-1}$ and
    * $\alpha_{i-1} = A_{i-1}\eta_{i-1}$ for some variable A_{i-1} and some suffix η_{i-1}.
  - Everything constructed here comes from a leftmost derivation of w in G, so there exists a production $A_{i-1} \to \beta_{i-1}$ in G, for some β_{i-1} (a string of terminals and/or variables).
  - The production $A_{i-1} \to \beta_{i-1}$ gives that $\delta(q, \varepsilon, A_{i-1})$ contains (q, β_{i-1}) so $(q, y_{i-1}, \alpha_{i-1}) \underset{P}{\vdash} (q, y_{i-1}, \beta_{i-1}\eta_{i-1})$.
  - By construction, $y_{i-1} \in \Sigma^*$.
  - As everything constructed here comes from a derivation of w in G, y_{i-1} and β_{i-1} must have a (possibly empty) prefix of matching terminals.
  - The leftmost variable in $\beta_{i-1}\eta_{i-1}$ is A_i, which begins α_i.
  - Match all of the terminals at the top of the stack against the terminals in the input y_{i-1} (using $\delta(q, a, a) = \{q, \varepsilon\}$ for all $a \in T$).
  - Consume all matching terminals to obtain $(q, y_{i-1}, \beta_{i-1}\eta_{i-1}) \overset{*}{\underset{P}{\vdash}} (q, y_i, \alpha_i)$
  - Thus $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, y_i, \alpha_i)$.

The proof in the other direction We have proved $L(G) \subseteq N(P)$. Now we prove $N(P) \subseteq L(G)$.
Let $w \in N(P)$ be arbitrary. To show that $w \in L(G)$, we will show something more general:

- For **any** variable A, if $(q, w, A) \overset{*}{\underset{P}{\vdash}} (q, \varepsilon, \varepsilon)$, then $A \overset{*}{\underset{G}{\Rightarrow}} w$.
- This is sufficient:
  - By definition $w \in N(P)$ means that $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, \varepsilon, \varepsilon)$.
  - Then applying the above implication, with $A = S$ gives us that $S \overset{*}{\underset{G}{\Rightarrow}} w$, in other words, $w \in L(G)$.

The proof is by induction on n, the length of the chosen computation $(q, w, A) \overset{n}{\underset{P}{\vdash}} (q, \varepsilon, \varepsilon)$:

- Base case ($n = 1; n = 0$ is impossible since $A \neq \varepsilon$):
  - In the construction of the PDA P, the only transitions defined for a non-terminal A at the top of the stack are ε-transitions corresponding to productions for A in G.
  - So our computation must take one of these ε-transitions as its first (and hence its only) step.

- As this ε-transition takes w to ε, therefore $w = \varepsilon$.
- Also $(q, \varepsilon) \in \delta(q, \varepsilon, A)$, so by construction there is a rule $A \to \varepsilon$ in G.
- Therefore $A \overset{*}{\underset{G}{\Rightarrow}} w$ as desired.

The inductive case $(n > 1)$: The inductive hypothesis is that for all variables A, if $(q, w, A) \overset{*}{\underset{P}{\vdash}} (q, \varepsilon, \varepsilon)$ in $< n$ steps, then $A \overset{*}{\underset{G}{\Rightarrow}} w$. Now consider an n-step computation $(q, w, A) \overset{*}{\underset{P}{\vdash}} (q, \varepsilon, \varepsilon)$.

- The first step in the computation must use a production of the form $A \to Y_1 Y_2 \cdots Y_k$ (where each $Y_i \in T \cup V$), since those are the only rules valid when A is on top of the stack (and the transition is an ε-transition). So $(q, w, A) \vdash (q, w, Y_1 \cdots Y_k)$.
- Decompose the computation from this time forward into phases:
  1. takes stack from $Y_1 \cdots Y_k$ to $Y_2 \cdots Y_k$
  2. takes stack from $Y_2 \cdots Y_k$ to $Y_3 \cdots Y_k$
  3. $\vdots$
  4. (k) takes stack from Y_k to ε
- There will be some first point when the stack contains just $Y_2 \cdots Y_k$, then some first point when the stack contains just $Y_3 \cdots Y_k$, and so on, until the first time it has just Y_k.
  (Each step removes at most one symbol from the stack, and we eventually empty the stack.)
- Consider when the stack contains $Y_2 \cdots Y_k$: we know that $(q, w, A) \underset{P}{\vdash} (q, w, Y_1 Y_2 \cdots Y_k) \overset{*}{\underset{P}{\vdash}} (q, w', Y_2 \cdots Y_k)$, for some suffix w' of w, where $w = w_1 w'$.
- Rewriting the last statement, we have $(q, w_1 w', Y_1 Y_2 \cdots Y_K) \overset{*}{\underset{P}{\vdash}} (q, w', Y_2 \cdots Y_k)$.
- Because we can delete the unread suffix w' of the input word without affecting the validity of the computation, we therefore have that $(q, w_1, Y_1 Y_2 \cdots Y_k) \overset{*}{\underset{P}{\vdash}} (q, \varepsilon, Y_2 \cdots Y_K)$.
- Moreover, since in the computation, we never touch the symbols of $Y_2 \cdots Y_k$ on the stack, we also have that $(q, w_1, Y_1) \underset{P}{\vdash} (q, \varepsilon, \varepsilon)$.
- If Y_1 is a terminal, then by the shape of the PDA, $(q, w_1, Y_1) \underset{P}{\vdash} (q, \varepsilon, \varepsilon)$ implies that $w_1 = Y_1$ is also a terminal, so that $Y_1 \overset{*}{\underset{G}{\Rightarrow}} w_1$ holds trivially.
- Otherwise Y_1 is a variable, and by construction, this computation wit-

nessing $(q, w_1, Y_1) \underset{P}{\vdash} (q, \varepsilon, \varepsilon)$ takes $< n$ steps, therefore $Y_1 \overset{*}{\underset{G}{\Rightarrow}} w_1$, by our inductive hypothesis.

- To summarize, whether Y_1 is a terminal or a variable, we have $Y_1 \overset{*}{\underset{G}{\Rightarrow}} w_1$.
- We then apply this argument to each subsequent Y_i, obtaining a derivation $Y_i \overset{*}{\underset{G}{\Rightarrow}} w_i$ for all i, with $w = w_1 w_2 \cdots w_k$.
- Finally, we have a derivation for w, by combining these together: $A \Rightarrow Y_1 \cdots Y_k \overset{*}{\underset{G}{\Rightarrow}} w_1 Y_2 \cdots Y_k \overset{*}{\underset{G}{\Rightarrow}} w_1 w_2 Y_3 \cdots Y_k \overset{*}{\underset{G}{\Rightarrow}} w_1 \cdots w_k = w$
- This shows that $A \overset{*}{\underset{G}{\Rightarrow}} w$, as desired.

Next, from a PDA to a CFG

- We have shown the easier case: for a CFG G with language $L(G) = L$, we can exhibit a PDA P such that $N(P) = L$.
- Now, given a PDA P (which accepts by empty stack), we must exhibit a grammar G such that $N(P) = L(G)$.
- This is harder.
  - Suppose that, in P,

$$(q, w, X) \overset{*}{\vdash} (p, \varepsilon, \varepsilon),$$

    for some $X \in \Gamma$.
  - How does that happen?

How do we consume one symbol from the stack?

- Suppose the first step in the computation is $(q, az, X) \vdash (r, z, Y_1 Y_2 \cdots Y_k)$, where $a \in T$ or $a = \varepsilon$.
- The process begins by removing X from the stack, replacing it with some string $Y_1 \cdots Y_k$, and moving to state r. ("phase 1" for this computation)
- Then, we go from state r to some state r_1, eventually removing all of the symbols of Y_1 from the stack, so that it consists of just $Y_2 \cdots Y_k$; this process may consume some input letters,from w, or it might not. ("phase 2")
- Eventually, we wind up in state p, in the instantaneous description $(p, \varepsilon, \varepsilon)$, having read all of the letters of w and all of the symbols in $Y_1 \cdots Y_k$.

Making a grammar from this

To construct a grammar which generates the same language that the PDA accepts, we need to construct our productions to mimic the action of the transitions in the PDA.

- Include a non-terminal (i.e. variable) $[qXp]$ for **every** stack symbol X and **every** pair of states q and p.
- A string generated by the non-terminal $[qXp]$ in G corresponds with a (possibly partial) computation in the PDA. It will capture the input letters consumed so far and the current stack contents.
- We want: $[qXp] \overset{*}{\underset{G}{\Rightarrow}} w$ if and only if $(q, w, X) \overset{*}{\underset{P}{\vdash}} (p, \varepsilon, \varepsilon)$.
- Now we need to construct the productions of the grammar to mimic the transitions in P.
- Examine each rule of the transition function, one at a time.
- Suppose that $\delta(q, a, X)$ contains $(r, Y_1Y_2 \cdots Y_k)$. **Note**: a can be ε; so can $Y_1 \cdots Y_k$.
  - If the output pair is (r, ε), so that $k = 0$, then add a production $[qXr] \to a$.
    - Read a, pop X, move to state r.
  - Otherwise, if $k > 0$, then add productions $[qXr_k] \to a[rY_1r_1][r_1Y_2r_2] \cdots [r_{k-1}Y_kr_k]$ to the grammar, for **all** choices of $r_1, \ldots, r_k$ from our state set Q.
    - Read a, pop X, push $Y_1 \cdots Y_k$ onto the stack, allowing for **any** sequence $r_1, \ldots, r_k$ of states to be used to pop $Y_1 \cdots Y_k$, in the end, move to state r.
- The language of the PDA is the union of the language corresponding to emptying the stack in **any** possible state, so for **every** state p we add a rule: $S \to [q_0Z_0p]$.
- Then by construction $(q_0, w, Z_0) \overset{*}{\vdash} (p, \varepsilon, \varepsilon)$ if and only if $S \Rightarrow [q_0Z_0p] \overset{*}{\Rightarrow} w$, as we had wanted.

How does this work? The basic idea, again: Each PDA transition

- either consumes a symbol from the input, or does not,
- changes state and
- takes one symbol off the stack, and then puts some new symbols on the stack, or does not.

To mimic this in one step of a leftmost derivation in the grammar which we have just constructed:

- Look at the right hand side of a particular production.
- The terminal (or ε) at the beginning of the right hand side is exactly what we expect to read from the input, according to the transition function in the given PDA.
- Then the string of nonterminals, each of the form $[rY_1r_1]$ or $[r_{i-1}Y_ir_i]$ indicate, in the Y_i characters, the symbols that get pushed onto the

stack.

- The first state in the first non-terminal (namely r) is the state that the PDA goes into.
- The following states correspond to the states the automaton will be in when we have shrunk the stack.

The proof Actually, we are not going to do it.

- The basic structure is not especially interesting; it is in Section 6.3.2 of the text.
- The really interesting thing is the idea; the proof is just about filling in the details.

One thing to note: this construction is finite-size.

- This may not be obvious, but it is because the strings put on the stack are always of finite length. This means we add a finite number of rules, though potentially enormous. (How big?)

14 Lecture 14

Outline

1. Deterministic PDAs - M6 57-68

14.1 Deterministic PDAs

A PDA is non-deterministic by its nature. A **deterministic PDA (DPDA)** is a PDA, with extra constraints to make its processing deterministic.

- Must be at most 1 transition from every state, and for every (input letter, stack letter) pair.
- We can have ε-transitions, but each must be the **only** valid transition for the state and stack symbol.
- If there is a transition from q_0 with X on the top of the stack, that does consume an input letter, then there are no ε-transitions for (q_0, X).

More formally, for a given state q_0:

- $|\delta(q_0, a, X)| \leq 1$ always ($a \in T$, or $a = \varepsilon$),
- and if $|\delta(q_0, a, X)| = 1$, where $a \in T$, then $|\delta(q_0, \varepsilon, X)| = 0$.

Motivation: i.e. why do we care about DPDAs?

- **A:** Some CFLs are **not** accepted by any DPDA. I.e. the class of CFLs is strictly larger the class of DCFLs (a DCFL is the language of some DPDA).

Important difference versus DFAs:

- DPDAs can crash (if input symbols remain and no outgoing transition of the right shape is available, just as in a PDA); DFAs cannot crash.
- **Convention:** For now, assume that our DPDAs accept by final state.
- I won't spent time in class on the difference, for DPDAs, between final state and empty stack. See the additional notes about M6 on the course website if you are interested in this difference.
- **A Possibly Surprising Fact About DPDA Acceptance:** Acceptance by empty stack is **different** from acceptance by final state.

An example of a language accepted by a DPDA Ordinary PDA for the language:

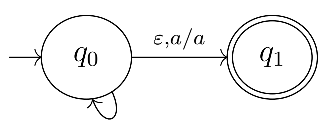
$$L = \{x \in \{a, b\}^* \mid n_a(x) > n_b(x)\}.$$

- (Words with more a's than b's.)

As we scan the word left-to-right, there will be more of one letter than the other.

- Keep track of the number of letters that we have more of.
- (Example: read in $aaaabab$ so far, then we should have three a's on the stack, since we have 5 a's and 2 b's.)
- Accept when there is still an a on the top of the stack at the end.

A nondeterministic PDA for L This gives a 2-state PDA like this:



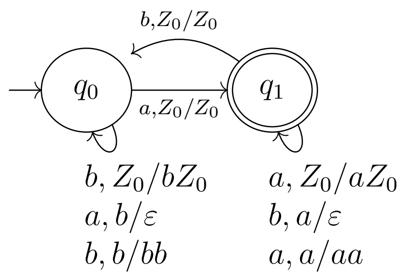
$a, Z_0/aZ_0$
$b, Z_0/bZ_0$
$a, b/\varepsilon$
$b, a/\varepsilon$
$a, a/aa$
$b, b/bb$

- First two transitions in state q_0: prefix is balanced
- Second two transitions in state q_0: reducing imbalance
- Last two transitions in state q_0: adding to imbalance
- Nondeterministic: ε-transition to state q_1.

Can we make this PDA deterministic? The current PDA is nondeterministic, because we have transitions when we pull an a off the stack that do eat input letters, and that do not.

Can we make small changes to make it deterministic?

- As before, but now with states
  - q_0 for "balanced or an excess of b's over a's" (initial, not final) and
  - q_1 for "an excess of a's over b's" (final).



This is a DPDA:

- There are no ε-transitions.
- There is always at most one choice for what to do next.
- In fact, there is exactly one choice:
  - There are two states.
  - There are two alphabet symbols.
  - By the shape of the machine, there are two possible top-of-stack symbols possible in each of q_0, q_1.
  - Thus there are $2^3 = 8$ transitions possible, each of which is defined in our picture.

But the stack still remembers the imbalance: we have just divided the transitions from before between the two states.
We jump from one state to the other when we switch from more a's than b's to not, or the other way around.

Not all CFLs are accepted by DPDAs

Fact (not proved in the text):
There exist context-free languages that are not accepted by DPDAs.

- Nondeterminism specifically makes PDAs more powerful.
- (This is not true for either Turing machines or for finite automata.)

Two different ways to prove this:

- Directly.
- Indirectly (Module 7)

We will talk a little about the first, but rely on the second.

Theorem 14.1.1. *Any regular language L is accepted by a DPDA.*

Proof. The DPDA just ignores the stack (i.e. every transition pops Z_0 and re-pushes it), and simulates the DFA that accepts L, which exists by Kleene's Theorem 5.1.1. □

An Example to show that DPDAs do not accept every CFL
We know that the language $L = \{x \in \{a, b\}^* | x \text{ is a palindrome}\}$ is a CFL, and hence accepted by a PDA.
We can argue that it is not accepted by any DPDA.

- Why? There has to be a way to identify the palindrome's middle.
- After reading the first 8 letters of the string *aabaabaa*, must both be ready to accept, if that is the end of the string, and be ready to read in more letters, in case it is just the first half of the string.
- In a DPDA, that is just not possible.
- Especially bad - we need to keep track of lots of other possibilities: the 8-letter string is the beginning of the 11-letter palindrome *aabaabaabaa*, etc.
- We cannot keep track of all of these choices for the middle: only one transition available at a time.

This can be made into a proof if we know how many states the DPDA is claimed to have; try it.
DPDAs and ambiguity

Theorem 14.1.2. *If P is a DPDA, accepting by empty stack, then $N(P)$ has an unambiguous grammar.*

Proof.
- Follow the grammar produced in the theorem that transformed a PDA into a CFG.
- The only place of some concern is that transitions where $\delta(q, a, X)$ included multiple symbols, $(p, Y_1Y_2 \cdots Y_n)$ turned into a pile of distinct new rules in the grammar.
- However, since the PDA is deterministic, we can see that only one derivation in the grammar will actually lead to the word that is accepted by the DPDA.

□

You can modify the proof to work for acceptance by final state, too. (That is Theorem 6.21 in the text.)
Overall hierarchy of languages We now know that:

- All finite languages are regular.

- All regular languages can be accepted by a DPDA, so are DCFLs.
- There are non-regular DCFLs (we just saw that $L = \{x \in \{a,b\}^* \mid n_a(x) > n_b(x)\}$ is a DCFL - it is an exercise to prove that L is not regular).
- DCFLs are all not inherently ambiguous (Theorem 14.1.2).
- There are languages that are not accepted by DPDAs that are not inherently ambiguous (e.g. palindromes).
- Some CFLs that are not inherently ambiguous are not DCFLs (e.g. palindromes).
- All CFLs are accepted by PDAs, and the languages accepted by PDAs are exactly the CFLs.

15 Lecture 15

Outline

1. A Normal Form For CFGs - M7 1-19

15.1 A Normal Form For CFGs

Motivation: Introducing such a normal form will enable us (via Theorem 15.1.2) to develop a Pumping Lemma for CFLs.

Theorem 15.1.1. *For any CFG, G, there is a CFG, G', such that $L(G) = L(G')$ (with the possible exception of $\varepsilon \in L(G)$), where all productions of the grammar G' are of one of the following two forms:*

1. *$A \to BC$ where A, B and C are variables, or*
2. *$A \to a$, where A is a variable and a is a terminal.*

Grammars having only these productions are in ***Chomsky Normal Form****, or CNF.*

Theorem 15.1.2. *Let G be a CNF grammar. Let $w \in L(G)$ be arbitrary. Then* ***any*** *derivation of w in G takes $2|w| - 1$ steps.*

Proof. The proof is by induction on $|w|$. We will prove the stronger result: for any variable A in G, if $A \overset{*}{\Rightarrow} w$, then the derivation must be of length $2|w| - 1$ steps. This is sufficient because we may then take $A = S$.

- A CNF grammar cannot make ε, so the base case is $|w| = 1$.
- <u>Base ($|w| = 1$):</u>

  - I claim that the first (and only) step in the derivation is $A \to a$ ($w = a$, for some terminal a).
  - There are no nullable variables, so if we instead started with a rule of form $A \to BC$, we would have to produce ≥ 2 terminals in the end.
  - So the only derivation of a 1-letter word takes 1 step.
  - Since $1 = 2(1) - 1$, therefore the base case holds.
- Induction ($|w| > 1$):
  - The induction hypothesis is that for all words x satisfying $A \overset{*}{\Rightarrow} x$ and $|x| < |w|$, the derivation of x takes $2|x| - 1$ steps.
  - As we are not in the base case, the first step in the derivation of w must be of the form $A \to BC$.
  - Then we can write $w = w_B w_C$, where $B \overset{*}{\Rightarrow} w_B,\ C \overset{*}{\Rightarrow} w_C,\ |w_B| < |w|$ and $|w_C| < |w|$ as there are no nullable variables.
  - By the induction hypothesis, the derivations for w_B and w_C are of lengths $2|w_B| - 1$ and $2|w_C| - 1$, respectively.
  - So the overall derivation, first using the $A \to BC$ rule, and then the derivations for w_B and for w_C, takes $1 + (2|w_B| - 1) + (2|w_C| - 1) = 2(\underbrace{|w_B| + |w_C|}_{=|w|}) - 1 = 2|w| - 1$ steps.

□

Outline of the proof of Theorem 15.1.1 We need to replace many kinds of now-forbidden rules:

1. $A \to \varepsilon$
2. $A \to B$
3. $A \to BCD$ (or > 3 variables in the RHS)
4. (a) $A \to Bc$
   (b) $A \to bc$

Some of these simplifications are easier than others; none is especially hard.

1. **Removing ε rules**

   Defintion 15.1.3. *A variable A in a grammar G is **nullable** if $A \underset{G}{\overset{*}{\Rightarrow}} \varepsilon$.*

   If A is nullable, and there is a rule in the grammar $B \to AC$, we add a rule $B \to C$ to the grammar, and the language of the grammar will not change.

   - Previous derivation: $B \Rightarrow AC \overset{*}{\Rightarrow} C \cdots$

- New derivation $B \Rightarrow C \cdots$

We can do this for *any* nullable variable.

- The only word lost is ε, in the case where S is nullable.

Lemma 15.1.4. *To identify nullable variables, apply this test:*

(a) If there exists a rule $A \to \varepsilon$, then A is nullable.

(b) If there exists a rule $A \to B_1 B_2 \cdots B_m$, and all B_i are nullable, then A is nullable.

(c) No other variables are nullable.

Remark: We lose no generality by assuming that all the B_is are variables: If any B_i is a terminal, then the derivation cannot produce ε from A.

Proof. First, suppose that the test discovers the variable A. Then the shape of the test provides an explicit derivation $A \stackrel{*}{\Rightarrow} \varepsilon$, which witnesses that A is nullable.

Now we need to show that every nullable variable A is discovered by this test. Suppose that there is a k-step derivation $A \stackrel{k}{\Rightarrow} \varepsilon$. The argument is by induction on k, the length of the derivation.

- Base case ($k = 1$): Then there is a rule $A \to \varepsilon$, and so the above test discovers that A is nullable.
- Induction case ($k > 1$): The induction hypothesis is that for any strictly shorter derivation $B \stackrel{*}{\Rightarrow} \varepsilon$, the test discovers that B is nullable.
  - The first step in the derivation of ε from A is $A \Rightarrow B_1 \cdots B_m$, where all the derivations $B_i \stackrel{*}{\Rightarrow} \varepsilon$ have strictly fewer than k steps. (No terminals can occur in the first step, as the derivation ends in the empty word.)
  - So by the induction hypothesis, the test discovers the B_is are all nullable, and hence that A is nullable also.

$\square$

We now must add new rules to the grammar corresponding to the nullable variables.

- Suppose $A \to aBcD$, with B and D nullable.
- Add new rules: $A \to acD$, $A \to aBc$, and $A \to ac$ to the grammar, corresponding to the cases where B generates ε, where D does, and where both do.

In general: from a rule with m nullable variables on the right hand side, add $2^m - 1$ new rules, removing each possible subset of the list of nullable variables. (There are 2^m ways of including / excluding the m nullable variables, and we already have the original rule in which all m of them are included.)
Then, remove null productions $A \to \varepsilon$ from the grammar.

Theorem 15.1.5. *Let G_1 be the grammar constructed in this way from the original grammar G. Then either $L(G_1) = L(G)$ or $L(G_1) \cup \{\varepsilon\} = L(G)$.*

- We will not show both directions of the proof (See Theorem 7.9 in the text).
- Here is the proof that if $w \in L(G)$ and $w \neq \varepsilon$, then $w \in L(G_1)$ (i.e. a proof that $L(G) \setminus \{\varepsilon\} \subseteq L(G'))$.
- We will show more generally that, for **any** variable A, if $A \underset{G}{\overset{*}{\Rightarrow}} w$, then $A \underset{G_1}{\overset{*}{\Rightarrow}} w$.
- This is sufficient because we may then take $A = S$.
- The proof is by induction on k, the number of steps in the derivation $A \underset{G}{\overset{k}{\Rightarrow}} w$.
- <u>Base ($k = 1$):</u>
  - Then $A \to w$ is a production in G.
  - Since $w \neq \varepsilon$, therefore $A \to w$ is a production in G_1 also.
  - Therefore we have $A \underset{G_1}{\overset{*}{\Rightarrow}} w$, as required.
- <u>Induction ($k > 1$):</u> Suppose that we have a k-step derivation $A \underset{G}{\overset{k}{\Rightarrow}} w$, for $k > 1$.
  - The induction hypothesis is that for all derivations $B \underset{G}{\overset{\ell}{\Rightarrow}} x$ with $\ell < k$, we have $B \underset{G_1}{\overset{*}{\Rightarrow}} x$.
  - The first step in the derivation of w in G is $A \Rightarrow B_1 B_2 \cdots B_m$, where each B_i is a variable or a terminal.
  - At least one variable remains after the first step, as we are not in the base case.
  - Write $w = w_1 w_2 \cdots w_m$, where $B_i \underset{G}{\overset{*}{\Rightarrow}} w_i$ for all i. (If B_i is a terminal, say $B_i = w_i$, then $B_i \underset{G}{\overset{*}{\Rightarrow}} w_i$ trivially.)

- Some of the w_i may be ε, but not all, as $w \neq \varepsilon$. Let $C_1, \ldots, C_n$ be the B_i that correspond to the non-ε subwords of w.
- Since the other B_is are nullable, by construction there exists a derivation in G_1 that starts with $A \Rightarrow C_1 \cdots C_n$.
- Each C_i yields its corresponding w_i in G, in fewer than k steps.
- So, by induction, $C_i \overset{*}{\underset{G_1}{\Rightarrow}} w_i$, for all i. Then derive w in G_1 via

$$A \underset{G_1}{\Rightarrow} C_1 \cdots C_n \overset{*}{\underset{G_1}{\Rightarrow}} w_1 C_2 \cdots C_n \overset{*}{\underset{G_1}{\Rightarrow}} \cdots \overset{*}{\underset{G_1}{\Rightarrow}} w_1 \cdots w_n = w.$$

2. **One-variable transformations**

Defintion 15.1.6. *A **unit production** is a production of the form $S \to A$, with only one variable on the right hand side.*

We want to get rid of unit productions. Why?
- One reason: avoid cycles like $S \Rightarrow A \Rightarrow B \Rightarrow S \Rightarrow \cdots$.

Easy:
- Basic idea: find all of the variables we can get to from a given variable.
- If $S \overset{*}{\Rightarrow} A$, then add all of A's productions directly to S's productions.

Defintion 15.1.7. *Variables (A, B) are a **unit pair** if $A \overset{*}{\Rightarrow} B$.*

Theorem 15.1.8. *We can find unit pairs by a simple recursive definition:*
(a) (A, A) is a unit pair for any pair A.
(b) If (A, B) is a unit pair and there is a production $B \to C$ in our grammar, where C is a variable, then (A, C) is a unit pair.
(c) No other pairs are unit pairs.

Easy proof (another induction, which we will not do; it is Theorem 7.11 in the text) that this method finds all unit pairs.
Removing unit productions If $S \overset{*}{\Rightarrow} A$ in our grammar G, add the productions for A to the productions for S.
Then, remove all unit productions.
Denote the new grammar by G_1.

- Any production that previously used the derivation in G starting from $S \overset{*}{\Rightarrow} A \Rightarrow B_1B_2 \cdots B_m$ can now use the rule $S \to B_1B_2 \cdots B_m$ in the new grammar G_1.
- This shows that $L(G) \subseteq L(G_1)$.
- Now, consider a derivation of a word w in $L(G_1)$.
  - Suppose we use a rule $S \to B_1B_2 \cdots B_m$ in G_1 for a variable S that came from a rule $A \to B_1B_2 \cdots B_m$ in G, where (S, A) is a unit pair in G.
  - Take derivation $S \underset{G}{\overset{*}{\Rightarrow}} A \Rightarrow B_1B_2 \cdots B_m$.
  - Then the rest of derivation follows; any word we can derive in G_1, we can also derive in G.
- This shows that $L(G_1) \subseteq L(G)$.
- Therefore we have $L(G_1) = L(G)$.

3. **Remaining bad kinds of rules** For $A \to B_1B_2 \cdots B_m$, where $m > 2$, create a cascading sequence of rules:
   - Only two symbols on right hand side for each rule.
   - If we take the first rule for A, then we will produce (eventually) all of $B_1B_2 \cdots B_m$.

   This is not hard. Create $m-2$ new variables $C_1, \ldots, C_{m-2}$, and these rules:

$$
\begin{array}{rcl}
A & \to & B_1C_1 \\
C_1 & \to & B_2C_2 \\
C_2 & \to & B_3C_3 \\
 & \vdots & \\
C_{m-2} & \to & B_{m-1}B_m
\end{array}
$$

   The new derivation is: $A \Rightarrow B_1C_1 \Rightarrow B_1B_2C_2 \overset{*}{\Rightarrow} B_1B_2 \cdots B_m$.
   If some of the B_i are terminals, then some of the rules we have just added are still are not allowed in a CNF grammar.
   We will correct this in the next (and last) step.
4. **The last step** In Chomsky Normal Form, a grammar has two kinds of rules:
   - $A \to BC$, for variables A, B and C
   - $A \to a$, for variables A and terminals a

   If we start with an arbitrary grammar, and we:
   - Remove ε-productions

- Remove unit productions
- Remove long productions

then the only possible remaining obstacle to being in CNF is that we might still have rules of the form $A \to bc$ or $A \to Bc$, with one terminal on the right hand side of the arrow, but two symbols.

The last step, finished This is easy:

- For a rule of the form $A \to Bc$:
  - Add the variable: X_c.
  - Add the productions:
    - $A \to BX_c$
    - $X_c \to c$
  - Then remove $A \to Bc$.
- For a rule of the form $A \to bc$:
  - Add two new variables:
    - X_b, and
    - X_c.
  - Add three new productions:
    - $A \to X_b X_c$,
    - $X_b \to b$, and
    - $X_c \to c$.
  - Then remove $A \to bc$.

Throughout, we must make certain that the new variable names added are not duplicates of any existing variable names. Then since the new variables are only used in these derivations, so they do not change the language of the grammar.

The new grammar fits the desired framework.

Chomsky Normal Form algorithm From a general CFG:

1. Remove ε-productions.
   (a) Find nullable variables.
   (b) Change rules using them
   (c) Then remove all ε-productions.
2. Remove one-variable productions.
   (a) Find unit pairs (A, B) for each variable A.
   (b) Add B's rules to A.
   (c) Then remove one-variable productions
3. Remove long productions.
   (a) Create cascading sequence of definitions.
4. Remove terminals from two-letter rules.

(a) Create a new variable for each terminal, and substitute it into the rules

Example: Let

$$G : S \to AB, A \to \varepsilon|0, B \to 1.$$

Note that $L(G) = \{1, 01\}$. Construct G', in Chomsky Normal Form, such that $L(G') = L(G)$.

Solution:

1. nullable variables A is nullable. Hence to construct G_1 from G, we
   (a) add $S \to B$ and
   (b) remove $A \to \varepsilon$.

$$G_1 : S \to AB|B, A \to 0, B \to 1.$$

2. unit pairs (S, B) is a unit pair. Thus we
   (a) add $S \to 1$ and
   (b) remove $S \to B$.

$$G_2 : S \to AB|1, A \to 0, B \to 1.$$

   Observe, G_2 is already in CNF.
3. long productions No changes: $G_3 = G_2$.
4. terminals in two-letter productions No changes: $G' = G_4 = G_3$.

It is now clear that $L(G') = L(G) = \{1, 01\}$.

16 Lecture 16

Outline

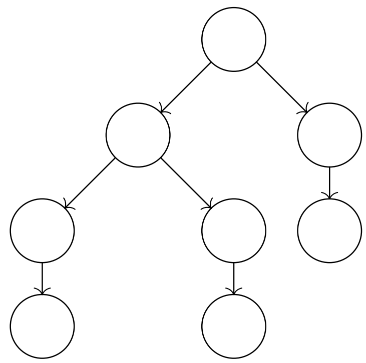
1. Pumping Lemma for CFLs - M7 20-37

16.1 Pumping Lemma for CFLs

Toward a pumping lemma for CFGs

Definition 16.1.1. *The* ***height*** *of a parse tree is the number of edges in a longest path from root to leaf.*

Example: The height of



is 3.

Remarks:

1. The parse tree for a CNF grammar is binary everywhere, except at the bottom of a branch, where a variable becomes a terminal.
2. The parse tree can be rooted at **any** variable, A, of the grammar G.

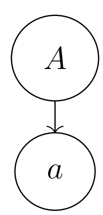
Suppose we have a CNF grammar, G, with p variables.

- Given k leaves, the height of the tree is **at least** $1 + \log_2 k$ (Proof coming soon).

Theorem 7.17: Suppose we have a parse tree according to a CNF grammar, G, and suppose the yield of the tree is some word w. If the height of the tree is ℓ, then $|w| \leq 2^{\ell-1}$.

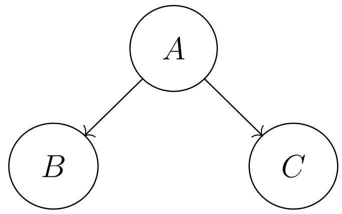
Proof.

- The proof is by induction on ℓ.
- Base ($\ell = 1$): (N.B. $\ell = 0$ is not possible: a parse tree for a word must at least turn its root variable into a terminal.)
  - The tree looks like

    i.e. only a root and a leaf.
  - Therefore $|w| = 1$, and $1 = |w| \leq 2^{\ell-1} = 2^0 = 1$ holds.
- Induction ($\ell > 1$):
  - The induction hypothesis is that any parse tree (for a word) of height $q < \ell$ has yield of length at most 2^{q-1}.

– As we are not in the base case, the tree must start with



The root of the tree must use a production of the form $A \to BC$.

– The **max** heights of the subtrees rooted at each of B and C are each $\ell - 1$.

– Thus the induction hypothesis applies to the subtrees rooted at B and C: these subtrees both have yields of lengths at most $2^{(\ell-1)-1} = 2^{\ell-2}$.

– The yield of the tree is the concatenation of the yields of these two subtrees, thus its length is at most $2^{\ell-2} + 2^{\ell-2} = 2\left(2^{\ell-2}\right) = 2^{\ell-1}$.

$\square$

Theorem 7.17 implies that the height of a parse tree for a word of length k is at least $1 + \log_2 k$.

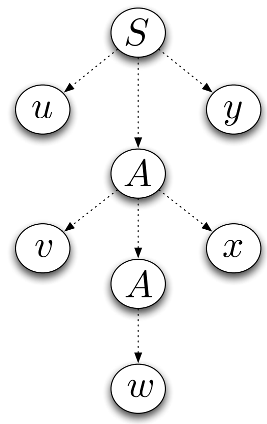
- By the Theorem we have $k \leq 2^{\ell-1}$, where ℓ is the height of a parse tree.
- Then we must have $\log_2 k \leq \ell - 1$, or in other words $\log_2 k + 1 \leq \ell$.

Theorem 7.17 further implies that, for a word $z \in L(G)$ with length at least 2^p (where p is the number of variables in G), a parse tree for z has height at least $p + 1$.

Repeated variables on the parse tree Assuming G has p variables, any parse tree of height $\geq p + 1$, must have a repeated variable (call it A) somewhere (by pigeonhole principle).

One derivation of the word z in G is of the form:

$$S \stackrel{*}{\Rightarrow} uAy \stackrel{*}{\Rightarrow} uvAxy \stackrel{*}{\Rightarrow} uvwxy = z.$$



Remarks:

1. In the above diagram, the dotted lines indicate derivations, of one or more steps.

Pumping In G:

- 0 repetitions

$$S \stackrel{*}{\Rightarrow} uAy \stackrel{*}{\Rightarrow} uwy = uv^0wx^0y$$

- 1 repetition As above
- 2 repetitions

$$S \stackrel{*}{\Rightarrow} uAy \stackrel{*}{\Rightarrow} uvAxy \stackrel{*}{\Rightarrow} uvvAxxy \stackrel{*}{\Rightarrow} uv^2wx^2y$$

- i repetitions (any $i \geq 0$)

$$S \stackrel{*}{\Rightarrow} uv^iwx^iy$$

Making a pumping lemma

Important: $A \stackrel{*}{\Rightarrow} vAx$ **and** $A \stackrel{*}{\Rightarrow} w$.

So $A \stackrel{*}{\Rightarrow} vAx \stackrel{*}{\Rightarrow} vvAxx \stackrel{*}{\Rightarrow} v^iAx^i \stackrel{*}{\Rightarrow} v^iwx^i$, for any choice of $i \geq 0$!

This will give our pumping lemma for CFLs.

Note: it cannot be the case that $vx = \varepsilon$, as a **non-trivial** repetition of A occurs, and unit productions $A \stackrel{*}{\Rightarrow} A$ are not allowed in a CNF grammar.

Q & A

1. What does "**non-trivial** repetition of A" mean?
   A: In the above parse tree, the two As are **not** the same node, i.e. ≥ 1 derivation steps (i.e. ≥ 1 levels om the height of the tree) must separate the two As.

Last Trick: Choose a repeated variable A **as close to the bottom of the parse tree as possible**.

- By assumption $|z| \geq 2^p$, so that a parse tree for z has height at least $p+1$.
- There exists a terminal in z with a path of length at least $p+1$ above it.
- By pigeonhole principle, there is a (non-trivially) repeated variable (say A) in this path, no more than $p+1$ levels above the leaf.
- Then we have $A \overset{*}{\Rightarrow} vAx$ and $A \overset{*}{\Rightarrow} w$, i.e.
  - the yield of the subtree rooted at the lowest A is w, and
  - the yield of the subtree rooted at the second lowest A is vwx.
- By construction the subtree rooted at the second lowest A has height at most $p+1$.
- Applying Theorem 7.17: $|vwx| \leq 2^{(p+1)-1} = 2^p$.
- As the repetition of A is non-trivial, therefore v and x are not both ε (unit productions $A \overset{*}{\Rightarrow} A$ are not allowed in a CNF grammar).

CFL Pumping Lemma (First Version)

Lemma: Let G be a CFG in Chomsky Normal form, with p variables.

- Any word $z \in L(G)$ of length at least 2^p can be decomposed as $z = uvwxy$, where
- $|vwx| \leq 2^p$,
- v and x are not both ε, and
- and for all nonnegative i, $uv^iwx^iy \in L(G)$.

As with the pumping lemma for regular languages, we can remove the dependency on a specific choice of CFG for the CFL, since all CFLs have a CNF grammar.

Pumping Lemma for CFLs (the way we will use it)

Pumping Lemma for CFLs 16.1.2. *Let L be a language. Suppose that for any $n > 0$, there exists a word $z \in L$ with $|z| \geq n$ such that:*

- *for **any** decomposition $z = uvwxy$, where $|vwx| \leq n$ and $|vx| > 0$, and*
- *there exists an $i \geq 0$ such that that $uv^iwx^iy \notin L$.*

*Then L is **not** context free.*

How To Use The CFL Pumping Lemma To Prove A Language L Is Not Context-Free

- For an arbitrary $n > 0$, find a word $z \in L$, at least n letters long.
- Look at all decompositions $z = uvwxy$, with $|vwx| \leq n, vx \neq \varepsilon$.

- Say something useful about the decompositions.
- For each decomposition, find an i such that uv^iwx^iy is not in L.
- Then the language L is not context free.

Examples:

1. $L = \{a^ib^ic^i \mid i \geq 0\}$
   I claim that the language $L = \{a^ib^ic^i \mid i \geq 0\}$ is **not context-free**.
   - Let $n > 0$ be arbitrary.
     Let $z = a^nb^nc^n$. Clearly $z \in L$.
   - Consider decompositions $z = uvwxy$ with $|vwx| \leq n$ and $vx \neq \varepsilon$.
   - Say something useful about all such decompositions.
     - All such decompositions have one or two types of letters in vwx, but not all 3.
     - (Why? The smallest consecutive substring with all 3 symbols is ab^nc; it has length $n + 2$.)
     - In particular, vx omits one or two letters of the set $\{a, b, c\}$.
   - For each decomposition, find an i such that uv^iwx^iy is not in L.
     - Consider uwy (i.e. take $i = 0$). Observe that uwy does not have the same number of a's, b's and c's, since one of these letters is not in vx, and at least one is!
     - Hence, $uwy = uv^0wx^0y$ is not in L. (Neither is $uvvwxxy$).

   We have shown that z cannot be pumped, and hence, L is not context free.
2. Consider $L = \{a^ib^jc^k \mid i < j, i < k\}$.
   - Let $n > 0$ be arbitrary.
   - Long word: $z = a^nb^{n+1}c^{n+1} \in L$.
   - Consider decompositions $z = uvwxy$ with $|vwx| \leq n$ and $vx \neq \varepsilon$.
     In all of them, vx has either no a's, or has a's but no c's.
     - Case 1: No a's.
       Then uwy has fewer b's or fewer c's than z, but there are not fewer a's.
       So uwy does not have fewer a's than both b's and c's, and therefore uwy is not in L.
     - Case 2: a's, but no c's.
       $uvvwxxy$ has as at least as many a's as c's, so $uvvwxxy$ is not in L.
   - So **no** decomposition of our long word $z = a^nb^{n+1}c^{n+1}$ can be pumped.

   And, thus, L is not context free.

3. Somewhat surprising, maybe:
   $L = \{ss \mid s \in \{a, b\}^*\}$.
   L includes words like aa or $abbabb$ or ε or $abaaba$.
   - For a given $n > 0$, find a long word. We will use $z = a^n b^n a^n b^n$. (This choice might not be so obvious.)
   - Decompose into $z = uvwxy$, with $|vwx| \le n$ and $vx \ne \varepsilon$. Then uwy must have at least one a or one b removed from one of the two copies of the identical string.
   - But when we remove vx from $uvwxy$ to form uwy, and lose a letter from the copied word, we cannot lose the corresponding letter on the other side; it is too far away.
   - Therefore $uwy \notin L$.
   - So L is not context-free.

   (See Example 7.21 of the text for all the gory details.)
4. **A bit surprising**
   The very similar-looking $L = \{ss^R \mid s \in \{a, b\}^*\}$, of even-length palindromes, is context free, with this grammar:
   - $S \to aSa \mid bSb \mid \varepsilon$

   However the previous example is still not context-free; we cannot keep all the information available whenever it is needed. (PDAs, which only recognize CFLs, have trouble with doing this.)

17 Lecture 17

Outline

1. Closure Rules for CFLs - M7 38-45
2. Quick Review of Decidability/Undecidability - CS 245
3. Decision Problems for CFLs - M7 46-50

17.1 Closure Rules for CFLs

CFLs are:

1. Closed under union, concatenation, Kleene star and reversal.
2. **Not** closed under intersection or complementation.

The easy ones

1. Union:
   - Grammar $G_1 : S_1 \to \cdots$

- Grammar $G_2 : S_2 \to \cdots$ (with all different variable names)
- Construct: $G : S \to S_1 | S_2 \cdots$

2. Concatenation:
   - Grammar $G_1 : S_1 \to \cdots$
   - Grammar $G_2 : S_2 \to \cdots$ (with all different variables)
   - Construct: $G : S \to S_1 S_2 \cdots$
3. Kleene star:
   - Grammar $G_1 : S_1 \to \cdots$
   - Construct: $G : S \to \varepsilon | S_1 S$
4. **Reversal** Reversal is not hard, either.
   - Given a grammar G, construct a new grammar G', by reversing the outputs of all of the productions in G.
   - For example, if G has a production $S \to XYZ$, then add the rule $S \to ZYX$ to G'.
   - (If the grammar is in CNF, this works especially easily.)
   - Now for a given derivation of a word $w \in L(G)$, apply the corresponding rules in G' to generate $w^R \in L(G')$.
   - Then by construction, $L(G') = L(G)^R$.
   - Then since $L(G)^R$ is the language of a context-free grammar, therefore $L(G)^R$ is a context-free language.
5. **Intersection** We have already seen a language that shows that the intersection of two CFLs is not always a CFL.
   $L = \{a^i b^i c^i \mid i \geq 0\}$ (we saw that this language is **not** context-free).
   - $L = L_1 \cap L_2$, where:
     - $L_1 = \{a^i b^i c^j \mid i, j \geq 0\}$.
     - $L_2 = \{a^i b^j c^j \mid i, j \geq 0\}$.
   - L_1 and L_2 are each the concatenation of two context-free languages, so context free.
   - In detail, define
     - $L_{11} = \{a^i b^i \mid i \geq 0\}$ (a CFL, with grammar $G : S \to aSb|\varepsilon$), and
     - $L_{12} = \{c^j \mid j \geq 0\} = L(c^*)$ (regular, and thus a CFL).
     - Then $L_1 = L_{11} L_{12}$.
     - And $L_{21} = \{a^i \mid i \geq 0\} = L(a^*)$ (regular, and thus a CFL).
     - $L_{22} = \{b^j c^j \mid j \geq 0\}$ (a CFL, with grammar $G : S \to bSc|\varepsilon$), and
     - Then $L_2 = L_{21} L_{22}$.

   Therefore, the class of context-free languages is **not** closed under intersection!

6. **Intersection of a CFL with a *regular* language**
   If L_1 is context free and L_2 is regular, then $L_1 \cap L_2$ is context free.
   - Suppose that a PDA M accepts L_1 by final state, and that a DFA D accepts L_2.
   - Let R be the states of M, and $F_M \subseteq R$ be the accept states.
   - Let S be the states of D, and $F_D \subseteq S$ be the accept states.
   - Define a PDA, P, which accepts by final state, with
     - States $Q = R \times S$,
     - Accept states $F = F_M \times F_D$,
     - and transition function δ, defined from the transition functions δ_M for M and δ_D for D (ignoring stack manipulations for the moment):

$$\delta(a,(r,s)) = \begin{cases} \{(\delta_M(\varepsilon,r),s)\} & \text{if } a=\varepsilon \\ \{(\delta_M(a,r),\delta_D(a,s))\} & \text{if } a \neq \varepsilon \end{cases}$$

     - Manipulate the stack in P exactly as it was manipulated in M.
   - Then P is a PDA, and from construction, we have that
     - P accepts w
     - if and only if M accepts w and D accepts w
     - if and only if $w \in L_1 \cap L_2$, so that $L_1 \cap L_2$ is a CFL.

   Note, this construction will **not** work for intersection of two arbitrary CFLs: both PDAs would need editing access to the one stack.
7. **Complementation**
   The following example will show that **the class of context-free languages is not closed under taking complements**. Let

$$\begin{aligned} L &= \{a^i b^i c^i \mid i \geq 0\} \\ L_1 &= \{a^i b^j c^k \mid i \neq j \text{ or } k \neq j\} \end{aligned}$$

   - Then L_1 is context-free, as it is the union of the four CFLs:
     i. $L_{11} = \{a^i b^j c^k \mid i < j\} = \{a^i b^j \mid i < j\}\{c^k \mid k \geq 0\}$,
     ii. $L_{12} = \{a^i b^j c^k \mid i > j\} = \{a^i b^j \mid i > j\}\{c^k \mid k \geq 0\}$,
     iii. $L_{13} = \{a^i b^j c^k \mid j < k\} = \{a^i \mid i \geq 0\}\{b^j c^k \mid j < k\}$, and
     iv. $L_{14} = \{a^i b^j c^k \mid j > k\} = \{a^i \mid i \geq 0\}\{b^j c^k \mid j > k\}$.
   - For example, a grammar for $\{a^i b^j \mid i < j\}$ is $G : S \to b|Sb|aSb$.

Now, consider $L_2 = L(a^*b^*c^*)'$. That is, L_2 is the set of words that are **not** of the form $a^ib^jc^k$, for any choice of i, j, k.

- Then L_2 is regular, as it is the complement of a regular language. (Exercise: What is a regular expression for L_2?)
- Then L_2 is a CFL.

Then $L_3 = L_1 \cup L_2$ is context-free, as it is the union of two context-free languages.

Note that words in L_3 are:

- of the form $a^ib^jc^k$ for some i, j, k, but not having $i = j = k$, or
- not of the form $a^ib^jc^k$, for any choice of i, j, k.

Now, consider L_3'. I claim that

$$L_3' = \{a^ib^ic^i \mid i \geq 0\}.$$

We have

$$\begin{array}{rcl} L_3' & = & (L_1 \cup L_2)' \\ & \underbrace{=}_{\text{DeMorgan}} & L_1' \cap L_2'. \end{array}$$

- $L_2' = (L(a^*b^*c^*)')' = L(a^*b^*c^*)$ is the set of words that **can** be written in the form $a^ib^jc^k$, for some choice of i, j, k,
- and L_1' is the set of such words for which $i = j = k$,
- and therefore our description of L_3' is correct.
- We have already seen that L_3' is **not** context free.

L_3 is context free, and its complement is **not** context-free.

Therefore the class of context-free languages is **not** closed under complementation.

Contrasts with DCFLs

DCFLs **are** closed under complementation.

- Proving this is non-trivial.
- See the additional notes for Module 7.

Simple proof that there are context-free languages that are not DCFLs: we just saw one. L is context-free, while L' is not context-free (and hence not a DCFL)

17.2 Quick Review of Decidability/Undecidability

Definition 17.2.1. *A **decision problem** is a problem which calls for an answer of either yes or no, on some input.*

Examples:

1. Given a program P and input I, will P halt when run with input I? (the **Halting Problem**)

Definition 17.2.2. *A decision problem is* ***decidable*** *if there exists an algorithm that, given an input to the problem,*

- *outputs yes (or true) if the input has answer yes, and*
- *outputs no (or false) if the input has answer no.*

A decision problem is ***undecidable*** *if it is not decidable.*

Remarks:

1. It is important to point out that the algorithm must always complete after **finitely many steps**.
2. To prove that a decision problem is decidable, **write down an algorithm to decide it**.
3. A decision problem may be undecidable, and yet have particular choices of input for which the correct yes/no answer can be determined. To say that a decision problem is undecidable is to say that no algorithm exists to give the correct yes/no answer **for every input**.
4. It is proved in CS 245 that **the Halting Problem is undecidable**. We will re-prove that fact soon, using Turing Machines.

17.3 Decision Problems for CFLs

Decision Problems for CFLs

Lots of the analogs to the problems we saw in Module 4 for regular languages are **not** solvable by computers.

What we can do: membership

1. Given a CFG, G, does $L(G)$ contain the word w?
   (a) If $w = \varepsilon$, then test the start variable, S, for nullability. If S is nullable, then yes; else no.
   (b) Otherwise, $w \neq \varepsilon$. The rest of the algorithm is to handle the case $w \neq \varepsilon$.
   (c) Turn the provided CFG into CNF.
   (d) Try all derivations of length $2|w| - 1$.
   (e) If any of them derives w, then yes; else no.
2. Given a PDA, P, does $N(P)$ contain the word w?
   (a) Turn P into a CFG (algorithm in slides).

(b) Use algorithm #1 on the constructed CFG. (Remark: Algorithm #2 is an example of a **reduction**. We will say more about reductions in M9.)
(c) It is tempting to just run the PDA on w. However, it might run forever, providing no answer.

3. Given a CFG, is its language empty?
   (a) First turn the CFG, G, it into CNF.
   (b) Apply:

   Theorem 17.3.1. *If a CFG in CNF, G, having p variables generates any words, then it must generate a word with $< 2^p$ letters.*

   Proof. i. Assume $L(G) \neq \emptyset$.
   ii. Let $z_0 \in L(G)$ be arbitrary.
   iii. If $|z_0| < 2^p$, then we are finished.
   iv. Otherwise, $|z_0| \geq 2^p$ and by the proof of the Pumping Lemma, we can decompose $z_0 = u_0v_0w_0x_0y_0$, with $|v_0w_0x_0| \leq 2^p$, $v_0x_0 \neq \varepsilon$ and $u_0w_0y_0 \in L(G)$.
   v. Let $z_1 = u_0w_0y_0$.
   vi. If $|z_1| < 2^p$, then we are finished.
   vii. Otherwise, $|z_1| \geq 2^p$ and $z_1 \in L(G)$ and so by the proof of the Pumping Lemma, we can decompose $z_1 = u_1v_1w_1x_1y_1$, with $|v_1w_1x_1| \leq 2^p$, $v_1x_1 \neq \varepsilon$ and $u_1w_1y_1 \in L(G)$. Let $z_2 = u_1w_1y_1$.
   viii. Continuing in this way we obtain a sequence of words in $L(G)$ having strictly decreasing lengths: $z_0, z_1, z_2, \ldots$.
   ix. As z_0 has finite length, after at most $|z_0| - 2^p + 1$ steps, we will obtain a word in $L(G)$ with length $< 2^p$.

   □

   (c) Enumerate **all** of them, and test membership for each, using algorithm #1.
   (d) If one word is in $L(G)$ then yes; else no.

Undecidable problems Other sensible problems are **undecidable**:

1. Given two CFGs, do their languages have any words in common?
2. Given two CFGs, are their languages equal?
3. Is the language of a CFG equal to Σ^*?
4. Given two CFGs, is the language of one a subset of the other's?
5. Is a given CFG ambiguous? (**Note**: this is about the **grammar**, not the **language**.)
6. Is a given CFL **inherently** ambiguous?

18 Lecture 18

Outline

1. Limits of Computation - M8 1-8
2. Introduction to Turing Machines - M8 9-12
3. Formal Definition of a Turing Machine - M8 13-27

18.1 Limits of Computation

1. In CS 245, you saw that **the Halting Problem is undecidable**.
2. In the slides, there are examples of
   (a) Two programs about which we can ask: does the program halt and return the output 1?, and for which
      i. one program clearly does while
      ii. the second program runs forever.
   (b) a variation on the proof from CS 245 about the Halting Problem.

Moral: Not every decision problem can be answered by a computer (i.e. is decidable).

18.2 Introduction to Turing machines

1. We have been studying simple models of computation.
2. We need a model of computation which is more similar to our understanding of how people (and computers) work.
3. In particular, we need to be able to access memory more robustly than we can in a PDA.

How do computers compute?

A simpler question: how do people compute?

1. We pull information from long-term memory into short term.
2. We work with the contents of short-term memory.
3. When finished, we write results to the long-term memory,
4. Repeat the above steps, until we have solved our problem.

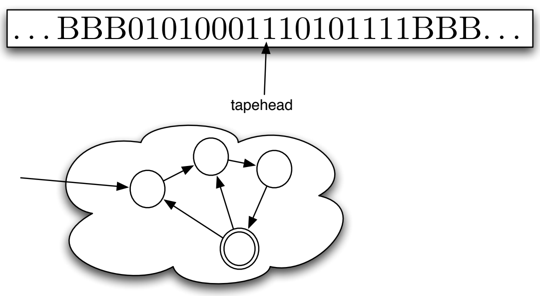
As an automaton model

Turing's model of a computing machine has two parts:

1. A finite automaton
   (a) has short-term memory, in its states
   (b) tells us how how to process the short-term memory, plus how to read from, and what to write to, long-term memory

2. A one-dimensional tape, representing long-term memory
    (a) A TM does not access memory as a real computer does, but they are equivalent in terms of computing power.

Church/Turing thesis: Anything we can do with a Turing machine we can do with any other reasonable computing model.
"If an algorithm exists, then a TM can be built to carry out that algorithm."
Turing machine = Finite automaton plus memory tape



- Long-term memory is only accessed at the tape head: we can only see one letter of memory at a time.

1 step in the TM:
- Given the current state in the FA, and the letter at the tape head
    - Move to a different state
    - Maybe change the letter at the tape head (or leave it alone)
    - Move tape head one cell left or right

18.3 Formal Definition of a Turing Machine

Formalization
To describe this, we need a 7-tuple:
1. Finite automaton control:
    (a) Σ: the alphabet for candidate words
    (b) Q: finite set of states
    (c) F: the subset of final states
    (d) q_0: initial state
2. Tape control:
    (a) Γ, the tape alphabet ($\Sigma \subseteq \Gamma$, since we start with $w \in \Sigma^*$ on the tape)
    (b) B, the blank symbol for the tape
        i. Note: $B \in \Gamma$, but $B \notin \Sigma$.

3. And a transition function:
   (a) δ: transition function. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$
      i. If $\delta(q, a) = (p, b, L)$ (for $a, b \in \Gamma$), then the machine
         A. switches states from q to p,
         B. overwrite a with b on the tape, and
         C. moves the tape head to the **left**.
      ii. This δ need not be a full function: if there is no value of $\delta(q, a)$, the machine **crashes** in state q upon seeing the tape symbol a.

The machine **halts** when we enter a final state, or crash.
A TM can also run forever.
More specifics about TMs
- The machine's tape is infinite in both directions.
- Our convention: We launch the TM with input $w \in \Sigma^*$ written on the tape ($|w|$ is always finite):
  – $|w|$ consecutive cells of the tape are filled with w's symbols,
  – the tape head points to w_1, the first symbol of w, and
  – all of the other cells of the tape are filled Bs.

Remember: the input could be of arbitrary size, but is always a finite string.

19 Lecture 19

Outline
1. Formal Definition of a Turing Machine - M8 13-27

19.1 Formal Definition of a Turing Machine

Instantaneous descriptions for Turing machines
Transitions in a TM After some number of steps of TM computation:
1. Current state, q.
2. The contents of the tape, $X_1 X_2 \cdots X_k$ (each $X_i \in \Gamma$; the rest of the tape is filled with Bs, infinitely many in both directions).
3. Tape head position: denote by underlining the current tape head position; the text notation is slightly different here - use our notation, because it is better.
4. If we want the tape head pointing to the first symbol, w_1, of a sequence of symbols, $w = w_1 w_2 \cdots w_n$, we will underline w, to indicate that the

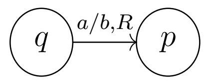
tape head points to the first symbol in it (e.g. if $w =$ "$carrot$", then $B\underline{w}B$ means $B\underline{c}arrotB$).

At the beginning of the execution in the TM, the instantaneous description is written as $(q_0, \underline{w_1}w_2\cdots w_n)$, or simply as $(q_0, \underline{w})$ (only write B when needed).

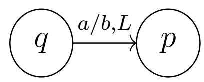
How to transition in the TM (where $|a| = |b| = 1$)

If the current instantaneous description of the machine is $(q, x\underline{a}y)$:

- If $\delta(q, a) = (p, b, R)$, then the new instantaneous description is $(p, xb\underline{y})$. Shorthand: $(q, x\underline{a}y) \vdash (p, xb\underline{y})$.



- If $\delta(q, a) = (p, b, L)$, then the new instantaneous description of the machine is $(p, x_1\cdots x_{k-1}\underline{x_k}by)$, and we write $(q, x\underline{a}y) \vdash (p, x_1\cdots x_{k-1}\underline{x_k}by)$.



Remarks:

1. A basic TM is **deterministic** (at most 1 transition for a state, tape-symbol pair; no ε-transitions)
2. δ need not be defined for every state, tape-symbol pair. A TM will crash if no applicable transition is defined.

Special cases: (where $|a| = |b| = 1$)

1. Tape head at leftmost non-B character then move L: the tape head moves to a new B character, B: $(q, \underline{a}y) \vdash (p, \underline{B}by)$, if $\delta(q, a) = (p, b, L)$.
2. Tape head at rightmost non-B character then move R: similarly, the tape head moves to a new B character: $(q, x\underline{a}) \vdash (p, xb\underline{B})$, if $\delta(q, a) = (p, b, R)$.
3. Erasing the last letter on the tape. For example, if $\delta(q, b) = (p, B, L)$, then $(q, xa\underline{b}) \vdash (p, x\underline{a})$. (Again, only write B when needed.)

Multi-Step Computations: Write $(q, x\underline{a}y) \overset{*}{\underset{M}{\vdash}} (p, v\underline{b}z)$ to indicate that we can move from the first configuration to the second, in some finite number of steps.

- To keep the notation uncluttered, we only indicate the machine M when necessary.

Acceptance in the TM

Definition: The Turing machine M **accepts** $w = w_1w_2\cdots w_n \in \Sigma^*$ if and only if $(q_0, \underline{w_1}w_2 \cdots w_n) \overset{*}{\underset{M}{\vdash}} (p, x\underline{a}y)$ for any final state $p \in F$ and any string $xay \in \Gamma^*$.

Remarks:

1. The tape need not be empty, and
2. the input need not have been fully examined.

Now we can define the language of a Turing machine:

- The **language**, denoted $L(M)$ of the machine M, is the set of words $w \in \Sigma^*$ which M accepts.

Definitions:

1. A language L is **recursively enumerable** if it is the language of some TM.
2. A language L is **recursive (decidable)** if it is the language of some TM **which halts on every input**.

How a TM can fail to accept some input word, w

How does a TM **reject** an input?

- If the TM is in a non-final state q, pointing at tape symbol a, and $\delta(q, a)$ is not defined, then it crashes and rejects the word.
- This is called "Halt and reject".
- (Later, once we move beyond implementation details and into describing algorithms for Turing machines, we will simply make our algorithms explicitly crash when necessary.)

A Turing machine can also fail to accept an input word w by running forever while processing it.

- This is possible with a PDA, too, or even with an ε-NFA.

Running Forever I.e. the TM goes into an infinite loop, never entering a final state. Running forever will be a possibility which we will need to handle carefully from now until the end of the course.

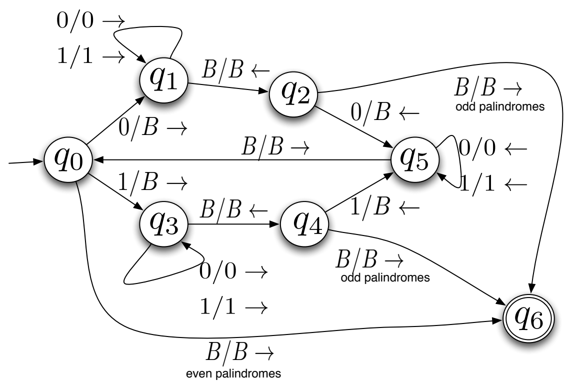
Halting means M either accepts w or crashes on w, we say that M **halts** on input w.

Not halting means M runs forever on w.

An example: palindromes over $\Sigma = \{0, 1\}$

Let's construct our first TM, to accept the language of palindromes, $L = \{x \mid x = x^R\}$.

Draw like a DFA, but with a new character for the tape and the arrow for the tape direction after a slash. (Or use L and R instead)



Explanation of the Diagram

1. Top Branch (q_1, q_2, q_5): Match a 0 at the start with a 0 at the end; delete both.
2. Bottom Branch (q_3, q_4, q_5): Match a 1 at the start with a 1 at the end; delete both.
3. Odd Length Palindromes:
   (a) If the last (middle) character is 0, then delete it as we move $q_0 \to q_1$. Then the tape is all B, so go $q_2 \to q_6$ and accept.
   (b) If the last (middle) character is 1, then delete it as we move $q_0 \to q_3$. Then the tape is all B, so go $q_4 \to q_6$ and accept.
4. Even Length Palindromes: We arrive back at q_0 after the last characters are deleted. Then the tape is all B, so go to q_6 and accept.
5. Rejecting All Non-Palindromes:
   (a) Start with 0 on the left; crash when we read a 1 in q_2.
   (b) Start with 1 on the left; crash when we read a 0 in q_4.

An accepting computation

Consider the palindrome $w = 010$.

$$
\begin{aligned}
(q_0, \underline{0}10) &\vdash (q_1, \underline{1}0) \\
&\vdash (q_1, 1\underline{0}) \\
&\vdash (q_1, 10\underline{B}) \\
&\vdash (q_2, 1\underline{0}) \\
&\vdash (q_5, \underline{1}) \\
&\vdash (q_5, \underline{B}1) \\
&\vdash (q_0, \underline{1}) \\
&\vdash (q_3, \underline{B}) \\
&\vdash (q_4, \underline{B}) \\
&\vdash (q_6, \underline{B})
\end{aligned}
$$

The machine accepts, having deleted the whole word.
A word not in L
Consider the non-palindrome $w = 0100$.

$$
\begin{aligned}
(q_0, \underline{0}100) &\vdash (q_1, \underline{1}00) \\
&\vdash (q_1, 1\underline{0}0) \\
&\vdash (q_1, 10\underline{0}) \\
&\vdash (q_1, 100\underline{B}) \\
&\vdash (q_2, 10\underline{0}) \\
&\vdash (q_5, 1\underline{0}) \\
&\vdash (q_5, \underline{1}0) \\
&\vdash (q_5, \underline{B}10) \\
&\vdash (q_0, \underline{1}0) \\
&\vdash (q_3, \underline{0}) \\
&\vdash (q_3, 0\underline{B}) \\
&\vdash (q_4, \underline{0})
\end{aligned}
$$

...and the machine crashes, rejecting $w = 0100$.
TMs can also accept non-context-free languages, like

$$L = \{s!s \mid s \in \{a, b\}^*\}.$$

We are not going to draw the machine that accepts L. (See §8.3.2 in the text.)
Idea: match letters from one copy of s with those in the other copy.

1. First, we ensure that there is exactly one ! character in the word.
2. Then, we match the first "remaining" character on each side of the ! character, and remove them.
3. Until there is nothing left.

20 Lecture 20

Outline

1. Programming Turing machines - M8 28-30
2. Computable functions - M8 31-37
3. Using Subroutines - M8 38-46

20.1 Programming Turing machines

1. Store a small (finite) amount of information in the state (e.g. in the palindrome TM, we had parallel state paths for reading 0 or 1 first from the tape).
2. Tag letters of the word with more bits of information, expanding the alphabet as needed.
3. Use subroutines where needed.

Remarks:

1. Our palindrome machine **decides** the palindrome language, since it accepts palindromes and explicitly crashes on all non-palindromes. Hence the palindrome language is recursive.
2. If our palindrome machine instead ran forever on at least one non-palindrome, it would only **accept** the language of palindromes.
3. If a Turing machine M accepts the language L, but does not decide L, then this does not imply that L is not recursive.
   (a) You may just have the wrong TM.
   (b) It might be possible to choose an improved TM which accepts L and halts on all inputs.

20.2 Computable functions

Idea: To construct a TM which computes a function, say $y = f(x)$, for some positive integers x and y:

1. Start with x on the tape, encoded as 1^x.
2. Finish by entering a final state, with the convention that output y will then be encoded on the tape as 1^y.

Definition 20.2.1. *A function $f(x)$, such that there exists a Turing machine to compute it as above, is called a **computable function**.*

Remarks:

1. Rather than treating 1^x as a word to be accepted or rejected, treat it as input to a function to be computed.
2. Our TM must halt and accept for **every** legal input 1^x, and put 1^y onto the tape as it does so.
3. Our assumption that we halt and accept for every input $x \in \Sigma^*$ was very strong. We can weaken this assumption and still get something useful.
4. Note that f need not be **total**: there could be words in Σ^* for which f is not defined. On those inputs, M may crash or run forever.
5. The function can have multiple arguments or be multi-valued.
6. Our first computable functions will be **unary** (i.e. they will be of the form $f : \Gamma^* \to \Gamma^*$). But many natural functions (e.g. addition, multiplication, expoentiation, etc. take two inputs, not just one.
7. If $f : \Gamma^* \times \Gamma^* \to \Gamma^*$, then we can separate the two arguments on the tape using B.

Examples:

1. $f(x) = x + 1$
   A TM to compute this f, will need to add another 1 at the end of the provided string of 1s on input.
2. $f(x) = 2x$
   A possible algorithm:
   (a) Mark the end of input, say with !.
   (b) For each 1 preceeding !
      i. Delete it, then,
      ii. Write two new 1s after the (output string) block of 1s following !
      iii. Repeat until a11 1s preceeding ! have been replaced with B.

(c) Delete !, move right to the first 1 of the output block.
(d) Halt and accept.

3. Consider the addition function: $+ : \mathbb{Z}^+ \times \mathbb{Z}^+ \to \mathbb{Z}^+,\ (x, y) \mapsto x + y$.
   - We encode the positive integer i on the tape as 1^i.
   - Then $x + y$ is the concatenation of the two arguments 1^x and 1^y, namely 1^{x+y}.
   - This $+$ is computable: we can certainly concatenate two strings with a Turing machine.

Exercise: Construct a Turing machine that performs this computation.

Characteristic functions

- In set theory, the **characteristic function of the subset** $X \subseteq S$ (for some "universe" S) is the function:

$$\begin{array}{rcccl} \chi_X & : & S & \to & \{0,1\} \\ & & s & \mapsto & \begin{cases} 1 & \text{if } s \in X \\ 0 & \text{if } s \notin X \end{cases} \end{array}$$

  which **indicates** whether an arbitrary $s \in S$ lies in X or not (like an **indicator function** in STAT 230).
- Hence the **characteristic function of the language** $L \subseteq \Sigma^*$ is the function:

$$\begin{array}{rcccl} \chi_L & : & \Sigma^* & \to & \{0,1\} \\ & & w & \mapsto & \begin{cases} 1 & \text{if } w \in L \\ 0 & \text{if } w \notin L \end{cases} \end{array}$$

- This function answers the question, "is $w \in L$?"
- **Key Observation:** membership in L is decidable if and only if χ_L is computable, and total.
- If the Turing machine M computes χ_L:
  - On any input $x \in \Sigma^*$, M **always** accepts.
  - When M halts, either 1 or 0 is left on the input, with the tape head pointing at that symbol.
- **If we can compute χ_L, then membership in L is decidable**
  - Suppose M computes χ_L.
  - Then we can build a machine M' to decide membership in L:

- On input x:
  - Since M always accepts, M' always gets through the M module.
  - Then M' either
    - accepts (if the tape head is pointing at a 1 when M accepts), so $x \in L$, or
    - crashes (if the tape head is pointing at a 0 when M accepts).
- Then M' decides membership in L.

- The other direction (showing how to compute χ_L assuming membership in L is decidable) is an exercise.

If f is computable (or respectively if χ_L is computable), then we say that f (or respectively L) **has an algorithm**.

20.3 Using Subroutines

How to use subroutines (assuming a subroutine TM is created and ready to use):

1. Set up the input for the subroutine.
2. Transition into the first state of the subroutine.
3. Wait until the subroutine halts.
4. Note that we must correctly handle the possibility that the subroutine runs forever (every subroutine is a Turing machine after all).
5. If we call a subroutine that runs forever, then our calling TM also runs forever. We can never do anything after the subroutine in this case. In our example, we assumed that M always halts, temporarily avoiding the issue.

We do **not** use subroutines in Turing machines to shorten the program code. The motivation is to improve clarity, not brevity. We are highly interested in the difference between what can be done in some finite number of steps (possibly large but still finite) versus what cannot be done in any finite number of steps.

Examples:

1. **Inserting a Character**

   The specification of our subroutine for inserting the character a at the current position of the tape head:

   - $(q_0, y\underline{z}) \overset{*}{\vdash} (q_F, ya\underline{z})$, where q_F is an accept state in the subroutine

machine and $a \in \Sigma$.
- Constraint: z does not have the blank character in it, so that we know when we have read the entire string z and moved it one character to the right.

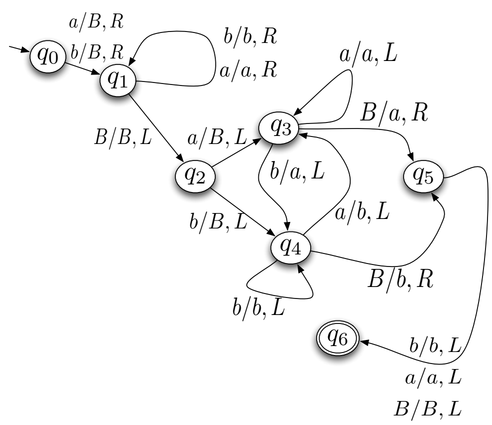
We might use this subroutine if we want to insert a different character into a string, by first inserting the character a, and then replacing the a with that character.

2. **Deleting a Character**

Delete the character the tape head is pointing at:

- $(q_0, y\underline{a}z) \overset{*}{\vdash} (q_F, y\underline{z})$, where q_F is again an accept state in the subroutine machine and $a \in \Sigma$.
- Again, constrain z to not contain any blank character.

Deletion machine



How it works

(a) In state q_0, we delete the current tape head character and move right.
(b) In state q_1, we move to the right until we have read the entire word on the input.
(c) Then, we remember the last letter, and put it into the position where the second-to-last letter was, remembering that.
(d) In state q_3, the previous symbol was a.
(e) In state q_4, the previous symbol was b.
(f) Push the whole way to the beginning of the string, and copy in

the last character as we move to state q_5.

(g) From state q_5, move the tape head back to the correct position.

(h) Accept in q_6.

Remark: If z were B, we would crash once we reach q_2. So clarify the specification to enforce $|z| \geq 1$.

Exercise: Enhance the provided TM so that it won't crash if z is B.

Example of Computation: processing $a\underline{b}ba$

$$\begin{aligned}(q_0, a\underline{b}ba) &\vdash (q_1, aB\underline{b}a)\\ &\vdash (q_1, aBb\underline{a})\\ &\vdash (q_1, aBba\underline{B})\\ &\vdash (q_2, aBb\underline{a})\\ &\vdash (q_3, aB\underline{b})\\ &\vdash (q_4, a\underline{B}a)\\ &\vdash (q_5, ab\underline{a})\\ &\vdash (q_6, a\underline{b}a)\end{aligned}$$

Storage in the state

Another trick, implicitly just used: augment a TM with a finite amount of memory.

In our previous example, we could combine q_2, q_3 and q_4 into a single state by storing the previous symbol seen in the memory of M.

21 Lecture 21

Outline

1. Variations on a Turing machine - M8 47-69
   (a) Multiple Tapes - M8 48-52
   (b) Multiple Tape Heads - M8 53-55
   (c) Non-Determinism - M8 56-68

21.1 Variations on a Turing machine

21.1.1 Multiple Tapes

In a **multi-tape** Turing machine, we have $k > 0$ tapes, each with its own tape head.

1. instantaneous description:
   (a) the state in the finite automaton,
   (b) the contents of each of the k tapes, and
   (c) the position of each of the k tape heads.
2. Transition function is of form: $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}^k$.
3. Conventions:
   (a) the input word is placed onto tape #1 (with tape head #1 pointing to its leftmost character), and
   (b) all other tapes start off filled with Bs (hence these tape heads positions do not matter).

Q: Is a multi-tape machine more powerful than an ordinary TM?

Theorem 21.1.1. *1. If a language is decidable by a multi-tape Turing machine, then it is decidable by a one-tape Turing machine.*
2. If a language is accepted by a multi-tape Turing machine, then it is accepted by a one-tape Turing machine.

Proof. We will focus on a language **accepted** by a 2-tape Turing machine, M, and show it is also accepted by a 1-tape Turing machine, M'. (If we can collapse 2 into 1, then inductively we can collapse any k into 1.) First, some housekeeping:

1. Suppose that M uses the tape alphabet Γ. (Recall that $B \in \Gamma$.)
2. Then let M''s tape alphabet be $\Gamma \times \{B, *\} \times \Gamma \times \{B, *\}$.
3. I.e. each position of the single tape for M' is two symbols from Γ, where each position is either tagged with a $*$ (to indicate where the tape head points), or not.
4. We may think of the single tape for M' as having four "tracks", to remember these four pieces of information.
5. For example, part of the tape might look like

B	0	1	0	B
B	*	B	B	B
B	1	1	B	B
B	B	*	B	B

Initial Setup: Prime tracks 1 and 2 of M's tape with the initial contents of M's tape,
Simulate one transition in the multi-tape TM M, in M' (Endow M' with some finite amount of memory).

1. Store M's current state in M''s finite memory.
2. Locate the two tape heads in M, by searching from the left of M''s tape, for the $*$ characters in the second and fourth positions of the four-tuple that is a letter in the tape alphabet of M'.
3. **Store the two corresponding characters** in M''s finite memory.
4. Using the known specification of M, determine the transition that M will take for the state, and pair of input tape symbols.
5. M' changes state, then updates the tape contents plus tape head positions appropriately.
6. Mark the new tape head positions.
7. Rewind the tape and repeat.
8. Let the final states of M' be the same as those for M, so that M' will accept exactly when M does.

$\square$

Why multi-tape TMs are useful:
Example: Consider $L = \{w!w \mid w \in \{0,1\}^*\}$.
1. We can create a one-tape TM to test membership in this language.
2. But it is easier to create a multi-tape TM.
3. See the details in the slides.

21.1.2 Multiple Tape Heads

1. We could have multiple tape heads on one tape.
2. Similarly to multiple tapes, we can simulate a multi-tape-head TM using an ordinary TM. **Moral:** A multi-tape-head TM is no more powerful than an ordinary TM.
3. See the details in the slides.
4. **Other memory alterations**
   (a) 2-dimensional memory (tape head moves up, down, left or right),
   (b) or even higher dimensions,
   (c) or a multi-tape machine where we store an address in the first tape that we can use to access the second tape: we can then model something like random-access memory, instead of sequential.

Example: Create a two-tape Turing Machine to recognize palindromes over $\{0,1\}$.
Solution: We will use
1. underlining, e.g. $\underline{a}$ to indicate the position of the first tape head, and

2. overlining, e.g. $\overline{a}$ to indicate the position of the second tape head.

By convention we start with tape contents

	B	$\underline{\overline{w_1}}$	w_2	$\cdots$	w_n	B	

where $w = w_1 \cdots w_n$. We have the following two cases, depending on the parity of n.

1. $\underline{n \text{ is odd:}}$ Write $n = 2k+1$, for some k. Then is clear that we can arrive at the tape contents

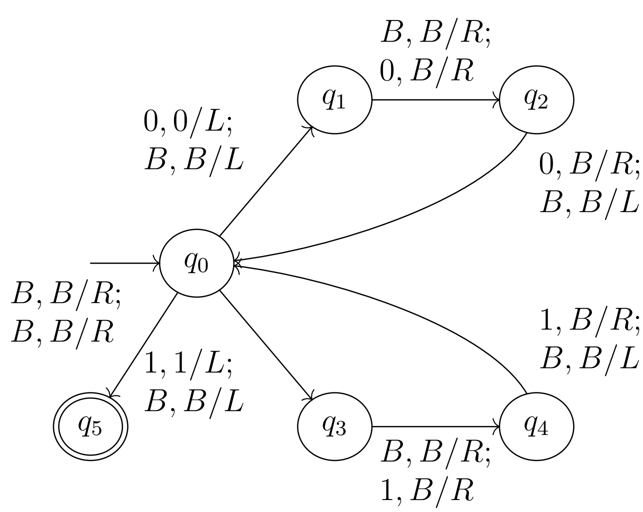
	B	$\underline{w_1}$	w_2	$\cdots$	$\overline{w_{2k+1}}$	B	

   and from here we can easily decide whether w is a palindrome or not by matching symbols on the left and right until we reach the middle of the word, and accept when the left tape head sees B. We won't show any additional details for this case.

2. $\underline{n \text{ is even:}}$ Write $n = 2k$, for some k. Then is clear that we can arrive at the tape contents

	B	$\underline{w_1}$	w_2	$\cdots$	w_{2k}	$\overline{B}$	B	

   and from here we need to demonstrate how we can decide whether w is a palindrome or not. We will still match symbols on the left and right until we reach the middle of the word. But we will need to move both tape heads to detect matched symbols, unlike in the previous case. The following Turing machine will carry out the matching algorithm in this case. We label each transition showing the actions at the two tape heads explicitly.



Exercise: Show the sequence of instantaneous configurations of this Turing machine while it processes:
(a) $w = 0110$
(b) $w = 0100$

21.1.3 Non-Determinism

A **nondeterministic** Turing machine can have multiple values for $\delta(q, a)$.

1. There can only be a **finite** number of entries in $\delta(q, a)$ (still no ε-transitions).
2. We write $(q, y\underline{x}) \vdash (p, w\underline{z})$ if **one** transition in $\delta(q, x)$ gets us to the second configuration. We do NOT attempt to describe all threads simultaneously.
3. The non-deterministic machine accepts input x if a valid computation exists that brings us from the starting configuration to an accepting configuration (i.e. accept iff at least one thread accepts).
4. It is **not** true that **all** threads must accept, or even halt.
5. A nondeterministic TM (NDTM) can only accept its language, not decide it.
   (a) What should it mean if one thread accepts while another crashes or runs forever?
6. NDTMs also can not compute functions.
   (a) What should it mean if two valid computations disagree about the value of $f(x)$?

Example: See the slides for a sketch of a two-tape head nondeterministic TM which accepts $L = \{ww \mid w \in \{a, b\}^*\}$.
How powerful is nondeterminism? If we have n steps in the computation, and 2 choices at each level, there could be 2^n branches! Are nondeterministic TMs more powerful?

- No.
- There is the possibility that they are faster (see the P *vs.* NP problem in CS 341).
- But we only care about what they **can** do, **not how quickly** they do it.

Theorem 21.1.2. *If L is accepted by a nondeterministic Turing machine M, then there exists a deterministic Turing machine M' that also accepts L.*

Proof. See the slides.

Key Idea: For each $n = 0, 1, 2, 3, 4, \ldots$: model all computations of n steps.

- if one computation accepts, then accept
- otherwise, increment n and repeat.

If one computation accepts, then we will eventually discover it. If not, then we will run forever. $\square$

Remark: Your text (Theorem 8.11) gives a very interesting proof that uses a queue, instead.

Also Useful: Slides 59-60 show that a TM with tape head moves $\{L, S, R\}$ (where S stands for "stay put") is no more powerful than an ordinary Turing machine, with only $\{L, R\}$.

22 Lecture 22

Outline

1. An Undecidable Language - M9 1-13
2. Other Undecidable Languages - M9 14-24

22.1 An Undecidable Language

Cantor proved $S = [0, 1] \subset \mathbb{R}$ is **uncountable**. See the proof in the slides. It uses a **diagonalization** argument. So will we, soon.

Set Cardinality

1. We say that infinite sets S and T have **equal cardinality** if there exists a **bijection** $f : S \to T$.
2. Recall that a function $f : S \to T$ is a **bijection** if it is both **injective** ("one-to-one") and **surjective** ("onto"), i.e. f is a 1-1 map from S to T.
3. In the case where S and T are both finite, then the definition of equal cardinality agrees with our intuition about finite sets: the sets are of equal cardinality iff they have the same number of elements.
4. However counter-intuitive things can happen in the infinite case: e.g., if
   (a) $S = \{\text{even integers}\}$, and
   (b) $T = \{\text{all integers}\}$, and
   (c) $f(x) = \frac{x}{2}$,
   then $f : S \to T$ is a bijection.

5. S and T are of equal cardinality, even though our intuition might tell us that S is "half" the size of T.

Countable Sets

1. A set S is **countably infinite**: S is of equal cardinality to $\mathbb{Z}$ (or to any infinite subset of $\mathbb{Z}$).
2. A set S is **countable**: S is finite our countably infinite.
3. A set S is **uncountable** if it is not countable. **Remark:** Uncountable is bigger than countable.

Countable sets exist, e.g. $\mathbb{Z}$, or { even integers }, etc. The positive integers are also countable. Cantor's proof that $[0, 1]$ is uncountable was the first rigourous proof that an uncountable set exists (upsetting to many, at the time).

Applying Cantor's Diagonalization to TMs

Idea: Produce a language which cannot be the language of any Turing Machine, i.e. it cannot be r.e..

1. We need to identify each Turing machine M (over the binary alphabet $\{0, 1\}$) with a binary string. See the technique in the slides. **Key Ideas:**
   (a) Number every ingredient of M's specification.
   (b) Define delimiters between a's ingredients, such that the delimiters do not overlap with a's strings.
   (c) Use a, b, to encode δ.
2. We encode a binary string to identify the TM, M; treat it as a natural number index for M. $f(M)$ is the natural number that we produce as above.
3. There can be multiple codes for $f(M)$ (but only finitely many). E.g. re-order the states.
4. **The set of Turing machines (hence the set of r.e. languages) is countable.** We just built a bijection to a subset of $\mathbb{Z}$.
5. There are uncountably many languages, even over a unary alphabet (proof in slides).
6. So there must exist a lot of (uncountably many) non-r.e. languages. But this does not tell us where to find them!

A language about Turing machines

Every Turing machine M has an identifier $w = f(M)$.

1. Consider the language

$$L_{SA} = \{w \mid \text{ the Turing machine } M \text{ represented by } w \text{ accepts } w\}.$$

Think $w = f(M)$, or its natural number equivalent. For M9, we can adopt the convention that TMs read and write natural numbers. If you prefer, use $\Sigma = \{0, 1\}$, and work with binary strings everywhere.

2. L_{SA} is the language of ids for TMs that accept when given their own ids as input.
3. SA indicates **self-accepting**.
4. It might seem weird to feed w into the machine identified by w. But lots of programs process other programs! E.g. parsers, compilers, interpreters, profilers, etc. Recall: $w = f(M)$ is just some natural number.

Fact: L_{SA} is not recursive, but we won't prove this yet.

An undecidable language

Instead, consider

$$L_{NSA} = \{w \mid \text{ the Turing machine } M \text{ represented by } w \text{ does } \textbf{not} \text{ accept } w\}.$$

1. L_{NSA} is the language of TM ids, w, such that the TM identified by w **does not** accept w.

Remarks:

1. Lots of natural numbers will not identify any TM.
2. Such a natural number is not a candidate for membership in L_{SA}, L_{NSA}, etc.
3. Checking whether an arbitrary natural number is $f(M)$, for some TM, M, is decidable.

Claim: L_{NSA} not the language of any Turing machine.

Proof.
1. We will prove this via an adaptation of Cantor's diagonalization argument.
2. Towards a contradiction, suppose $L_{NSA} = L(M)$, for some TM M. Let $w = f(M)$. Two cases:
   (a) If the Turing machine M represented by w accepts w (so that $w \in L(M)$), then $w \notin L_{NSA}$. But $L_{NSA} = L_M$, contradiction.
   (b) If the Turing machine M represented by w does not accept w (so that $w \notin L(M)$), then $w \in L_{NSA}$. But $L_{NSA} = L_M$, contradiction.
3. All possibilities lead to contradiction.
4. This contradiction shows that L_{NSA} differs from the languages of all Turing machines.

□

Remarks:

1. The claim shoes that L_{NSA} is **not recursively enumerable**.
2. It follows that L_{NSA} is also **not recursive** (Every recursive language is r.e.).
3. (The text denotes L_{NSA} as L_d, d for **diagonalization**.)
4. We now see why the uniqueness of the identifier for a given Turing machine is not crucial for this result:
5. **No** identifier w can represent a Turing machine M such that $L(M) = L_{NSA}$.

22.2 Other Undecidable Languages

The Universal Turing Machine
The Universal Turing Machine, U, simulates a Turing machine M, on an input w.

1. Input: a pair, (e, w).
2. If e does not represent any TM, then U rejects (e, w).
3. Othereise, if $e = f(M)$ for some Turing machine M, then
   (a) If M accepts w, then U accepts (e, w).
   (b) If M rejects w, then U rejects (e, w).
   (c) If M runs forever on w, then U runs forever on (e, w).

Does U exist?
A: Yes. U will have four tapes:

1. Keep (e, w), which really is (M, w)
2. Mimic M's tape
3. Maintain the current state, q, of M
4. Use for scratch work

To simulate one transition $(\delta(q, a))$ from M, inside of U:

1. Use tape #1, to look up M's transition $\delta(q, a)$.
   (a) Important: e on tape #1 is the full specification of M.
2. Update M's state on tape #3.
3. Update M's tape contents on tape #2; move the tape head L or R.
4. Repeat as needed.

Define $L_u = L(U)$: the **universal language**; it includes all pairs (e, w), where

1. $e = f(M)$ for some Turing machine M, and
2. $w \in L(M)$.

Key Observations:

1. The universal language, L_u, is **recursively enumerable**.

   (a) Trivial: U is a Turing machine; its language, L_u, is r.e.
2. L_u is not **recursive**.
   (a) Toward a contradiction, suppose that L_u is decidable, by a TM, U'.
   (b) Recall that L_{NSA} is not r.e., and thus not recursive.
   (c) We will obtain our contradiction by constructing a decider for L_{NSA}:
      i. Let e be an arbitrary candidate for membership in L_{NSA}, i.e. e is a specification for an arbitary TM.
      ii. Run U' on input (e, e).
      iii. If U' accepts (e, e), then reject.
      iv. If U' rejects (e, e), then
         A. If e is the encoding of a Turing machine (which we can test), then accept e.
         B. Else reject e.
      v. Since U' decides L_u, therefore U' cannot run forever.

   We still need to argue that M_{NSA} decides membership in L_{NSA}.
   (a) Suppose that e represents some Turing machine, M.
   (b) If M accepts e, then U' accepts (e, e). Hence we answer $e \notin L_{NSA}$.
   (c) If M does not accept e, then U' rejects (e, e). Hence we answer $e \in L_{NSA}$ (if e identifies a TM). If e does not identify a TM, then we answer $e \notin L_{NSA}$.

   This shows that M_{NSA} decides membership in L_{NSA}. But L_{NSA} is not recursive! This is not possible! Therefore U' cannot exist. This contradiction proves that L_u is not recursive.

Other recursively enumerable but not recursive languages

1. L_{SA}
   (a) L_{SA} is r.e.
      i. Construct a Turing machine M, which, on input w,
      ii. runs U on (w, w) (recall, U is the universal TM),
      iii. accepts if U accepts (w, w), and
      iv. rejects if U rejects (w, w).
      v. Note, U can run forever on input (w, w), causing M to run forever on input w.
      vi. Then by the definition of U, it is clear that $L(M) = L_{SA}$.
   (b) L_{SA} is not recursive.
      i. If it were, then we could decide membership in L_{NSA} easily:
      ii. Check $w \in L_{SA}$ and negate the answer for $w \in L_{NSA}$.

iii. but since L_{NSA} is not recursive (not even r.e.), this is impossible.

2. $L_{Halt} = \{(e, w) \mid \text{TM represented by } e\text{, halts when processing } w\}$ See M9 additional notes.

23 Lecture 23

Outline

1. Closure Rules for TM languages - M9 25-28
2. Reductions - M9 29-31
3. Other Undecidable Problems about Turing machines - M9 32-44
4. Rice's Theorem - M9 45-49

23.1 Closure Rules for TM languages

1. Complements
   Theorem: If L is recursive, then so is its complement L'.
   (a) Proof: See the slides.
   (b) Idea: Decide whether $w \in L$, then negate the answer for $w \in L'$.
   Remark: This result is **false** if we replace 'recursive' with 'recursively enumerable'. With respect to taking complements, r.e. languages are not as well-behaved as recursive languages.
2. **Theorem:** If both L and L' are recursively enumerable, then L is recursive.
   Proof:
   (a) We will construct a TM to decide L.
   (b) Suppose that M accepts L and M' accepts L'.
   (c) Create a 3-tape Turing machine, M_L, to simulate running M and M' in parallel, one step at a time.
      i. tape #1 controls which of M or M' is currently executing,
      ii. tape #2 simulates M's tape, and
      iii. tape #3 simulates M''s tape.
   (d) For any word w, either M or M' (and not both) must accept w in a finite number of steps.
      i. If M accepts w, then M_L accepts w.
      ii. If M' accepts w, then M_L rejects w.
   (e) Then M_L decides L.

□

The Theorem says that r.e. languages cannot be closed under complements. If they were, then the Theorem says that every r.e. language is recursive, which we know is not true, by last lecture.

3. Intersections

   Theorem: The intersection of two r.e. languages is r.e..

   (a) Proof: See the slides.
   (b) Idea: Check whether $w \in L_1$ and whether $w \in L_2$ (if so, both will be confirmed in finite time).

   An analogous result holds for recursive languages. The proof is even easier than for r.e. languages.

4. Unions

   Theorem: The union of two r.e. languages is r.e.:

   (a) Proof: See the slides.
   (b) Idea: Non-deterministically check in parallel whether $w \in L_1$ or whether $w \in L_2$ (if so, at least one will be confirmed in finite time).

23.2 Reductions

Proper definition of reduction

1. Suppose we have two decision problems P_1 and P_2.
2. Suppose also that we have an **algorithm** A, that transforms an instance for P_1 into an instance for P_2, such that:
   (a) "Yes" instances of P_1 get mapped to "yes" instances of P_2,
   (b) "No" instances of P_1 get mapped to "no" instances of P_2, and
   (c) the algorithm A always takes finite time.
3. Then A constitutes a **reduction** from P_1 to P_2.

Theorem 9.7: If there is a reduction from P_1 to P_2, then

1. If P_1 is undecidable, then P_2 is also undecidable.
2. If P_1 is non-recursively enumerable, then P_2 is also non-recursively enumerable.

Proof: See the slides.

23.3 Other Undecidable Problems about Turing machines

Instructor Note: You will only have time to present the details of at most one of these in class, I think. Present the details for L_{ne} and L_e only.

1. Empty language: Given a Turing machine M, does it accept $\emptyset$?
   (a) **Nonempty language:** Let

   $$L_{ne} = \{w \mid w \text{ identifies the Turing machine } M \text{ and } L(M) \neq \emptyset\}.$$

   i. Claim: L_{ne} is recursively enumerable.
   Here is an algorithm for a Turing machine M_{ne} that accepts L_{ne}:
      - Let w be the identifier for an arbitrary Turing machine M.
      - Non-deterministically guess a word x that M might accept (systematically test all possible words starting with the shortest ones first).
      - Nondeterministically execute M on all possible choices of x.
      - If M accepts any choice x, then M_{ne} accepts w.
      - Otherwise, all threads crash or run forever $\Rightarrow$ M_{ne} does not accept w.
      - Since M_{ne} accepts L_{ne}, therefore L_{ne} is r.e..

   ii. Claim: L_{ne} is **not recursive**.
   We give a proof using Theorem 9.7. (For a "bare hands" proof, see the slides.)
   Proof Using Theorem 9.7:
      - Define
        - P_1: Is (e, w) in L_u? (i.e. membership in the universal language, non-recursive), and
        - P_2: Is e' in L_{ne}? (i.e. the unknown thing)
      - The following algorithm A reduces P_1 to P_2:
        - Let (e, w) be an arbitrary instance for P_1.
        - Construct a new Turing machine, M', to do the following:
          * M''s input is an arbitrary $x \in \Sigma^*$.
          * M' will run U on (e, w).
          * If U accepts (e, w), then M' accepts x.

    - $*$ If U rejects (e, w), then M' rejects x.
    - $*$ U may also run forever on (e, w). Then M' runs forever on x.
  - Then we have

$$L(M') = \begin{cases} \Sigma^* & \text{if } U \text{ accepts } (e, w) \\ \emptyset & \text{otherwise} \end{cases}$$

  - Let e' represent M'.
  - Now take e' as the corresponding instance for P_2.
- Then "yes" instances of P_1 are sent to "yes" instances of P_2, and "no" instances of P_1 are sent to "no" instances of P_2.
- Thus A **reduces** membership in L_u, the universal language, to membership in L_{ne}.
- As L_u is undecidable, therefore by Theorem 9.7, so is L_{ne}.

iii. **Simpler notation from now on:** $L_{ne} = \{M \mid L(M) \neq \emptyset\}$.

- (From now on, we stop talking about f, the encoding, as much.)

(b) **Empty Language:** Let

$$L_e = \{M \mid L(M) = \emptyset\}.$$

Claim: L_e is not r.e. (and hence is not recursive).

- The complement of L_{ne} is

$$L'_{ne} = L_e \cup \{w \mid w \text{ is not the encoding of any Turing machine }\}.$$

- As $\{w \mid w$ is not the encoding of any Turing machine $\}$ is decidable, therefore it is also recursively enumerable.
- Towards a contradiction, suppose L_e is r.e..
- Then so is L'_{ne} (as it is the union of two r.e. languages), by r.e. language closure rules.
- But if a language and its complement are both r.e., then the language is decidable.
- And we know that L_{ne} is not decidable.
- So L_e cannot be r.e..

2. Finite language: Given a Turing machine M, is $L(M)$ finite?

(a) **Infinite language:** Let

$$L_\infty = \{M \mid L(M) \text{ is infinite }\}.$$

Then L_∞ is undecidable:
Proof: See Lecture Slides.

(b) **Finite language:** I claim that $L_{fin} = \{M \mid L(M) \text{ is finite }\}$ is also undecidable.
Proof: See Lecture Slides.

3. Regular language: Given a Turing machine M, is its language regular?
   (a) **Turing Machines with a Non-Regular Language**
      - Let $L_{nreg} = \{M \mid L(M) \text{ is not regular}\}$.
      - Then L_{nreg} is undecidable.
      - See Lecture Slides for the proof.

   (b) **Turing Machines with a Regular Language**
   I claim that $L_{reg} = \{M \mid L(M) \text{ is regular }\}$ is also undecidable.
   Proof: See Lecture Slides.

23.4 Rice's Theorem:

These results suggest that any "interesting" property of Turing machine languages is not decidable. We make this notion precise in the following Theorem.
Rice's Theorem: Let P be a property of some, but not all, recursively enumerable languages. (I.e. P is some non-empty proper subclass of the class of r.e. languages.) Then the language $L_P = \{M|L(M) \in P\}$ is undecidable.
Remark: Here we only work with the identifiers for legal Turing machines.
Proof: Cases

1. Suppose that the empty language, $\emptyset$, is not in P.
   - Let L be a non-empty recursively enumerable language that is in P.
   - Let M_L be a Turing machine that accepts L.
   - We will reduce membership in L_u to membership in L_P, then apply Theorem 9.7.
   - Let (M, w) be a candidate instance for L_u.
   - Create a new machine, M' that, on any input x, simulates U on (M, w).
     - If U accepts (M, w), then we simulate M_L on x.
       * If M_L accepts x, then M' accepts x.
       * If U rejects (M, w) or if M_L rejects x, then M' rejects x.
       * If U runs forever on (M, w), or if U accepts (M, w) and M_L then runs forever on x, then M' runs forever on x.

- The above construction gives us that the language of M' is

$$L(M') = \begin{cases} L & \text{if } U \text{ accepts } (M, w) \\ \emptyset & \text{otherwise} \end{cases}$$

- Let w' identify M'.
- Take w' as our candidate for membership in L_P.
- Then "yes" instances for L_u are sent to "yes" instances for L_P (as we chose $L \in P$ and $L \neq \emptyset$), and
- "no" instances of L_u are sent to "no" instances for L_P (as $\emptyset \notin P$).
- We have reduced membership in L_u to membership in L_P.
- But since L_u is undecidable, then by Theorem 9.7, L_P is also undecidable.

2. Suppose that the empty language, $\emptyset$, is in P.
   - Consider the property P', which is the negation of property P.
   - Then since $\emptyset \notin P'$, therefore testing membership in $L_{P'}$ is undecidable, by the proof of Case 1.
   - Since every Turing machine accepts a recursively enumerable language, therefore $(L_P)' = L_{P'}$.
   - For a contradiction, suppose that L_P is decidable.
   - Then $(L_P)' = L_{P'}$ is also decidable.
   - But this contradicts the fact that testing membership in $L_{P'}$ is undecidable.
   - Therefore L_P must be undecidable.

Decidable Problems About TMs

1. "Does this Turing machine have fewer than k states?"
2. "Does this Turing machine ever move the tape head left on any input?"
3. Most decidable problems are not interesting.

Problems About Two TMs

There are obvious problems about two languages, too: given M_1 and M_2,

1. is $L(M_1) = L(M_2)$?
2. is $L(M_1) \subseteq L(M_2)$?
3. is $L(M_1) \cap L(M_2) = \emptyset$, i.e. are the languages **disjoint**?

These are all undecidable, too.

1. Suppose we could decide any of these problems.
2. Then suppose we wanted to decide, for an arbitrary machine M, whether $M \in L_e$. (Recall: that problem is undecidable.)
3. Make new machines M_2 that rejects all inputs, and M_3 that accepts all inputs.

4. If $L(M) = L(M_2)$, then $M \in L_e$ (else $M \notin L_e$).
5. If $L(M) \subseteq L(M_2)$, then $M \in L_e$ (else $M \notin L_e$).
6. If $L(M) \cap L(M_3) = \emptyset$, then $M \in L_e$ (else $M \notin L_e$).

In all three cases, given a machine that decides the desired equality or containment, we could then decide membership in L_e, which is undecidable.

0

24 Lecture 24

Outline

1. Reductions, Revisited
2. Decision problems about CFGs and CFLs (Optional) - M9 50-59
   (a) Post's Correspondence Problem (Optional) - M9 51-53
3. Course Wrap-up - M9 60-69
4. Course Evaluations

24.1 Reductions, Revisited

Theorem 24.1.1. *If there is a reduction from P_1 to P_2, then*

1. If P_1 is undecidable, then P_2 is also undecidable.
2. If P_2 is decidable, then P_1 is also decidable.

Proof.

1. See the slides.
2. (a) Let A be a reduction from P_1 to P_2.
   (b) Assume that P_2 is decidable.
   (c) Let D be a decider for P_2.
   (d) The following algorithm decides P_1:
      i. Let x be an arbitrary instance for P_1.
      ii. Let y be the instance for P_2, which A produces from x.
      iii. Apply D to y, and return the same answer for x.
      iv. Because A is a reduction, the answer for y must also be correct for x.

□

Remarks:

1. The second statement is not really needed, as it is simply the contrapositive of the first. We state it explicitly because it can provide us with another means of proving that a decision problem is decidable.

2. The existence of a reduction from P_1 to P_2 can be thought of as asserting:
   (a) P_2 is at least as hard to solve as P_1, eqivalently
   (b) P_1 is no harder to solve than P_2.

   These facts were the starting point for Q2 on A06.

24.2 Decision problems about CFGs and CFLs (Optional)

We did not answer these questions, before:

1. CFG-intersection:
   (a) Given: Two grammars G_1 and G_2.
   (b) Question: Is $L(G_1) \cap L(G_2) \neq \emptyset$?
2. CFG-ambiguous:
   (a) Given: Grammar G.
   (b) Question: Is G ambiguous?

To show that they are both undecidable, we need to define a new problem.

24.2.1 Post's Correspondence Problem (Optional)

1. It is a funny undecidable game (sort of).
2. It dates to roughly WWII.
3. We are given a finite set of "tiles", where each tile contains two strings over a finite alphabet Σ.

$$(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n).$$

4. We want a non-empty string x, where it is possible to join together a sequence of tiles from the set (allowing repetition), and where the concatenation of the a_i strings and the concatenation of the b_i strings are both equal to x.

Example of PCP

Tiles:

$T_1 : (00, 001), T_2 : (11, 10)$ and $T_3 : (011, 1)$.

Want: string x to obtain from both parts of the tiles.

1. (**Note**: we do not know what x is!)
2. Guess x. Suppose $x = 001100011$.
3. Tile sequence: (T_1, T_2, T_1, T_3).

4. Look at the first strings: $00 + 11 + 00 + 011 = 001100011$
5. And the second strings: $001 + 10 + 001 + 1 = 001100011$
6. These are the same.
7. We have exhibited a solution to this instance of PCP.
8. There are instances of PCP for which no solution exists. See Example 9.14 on p402 of the text.
9. So the question naturally arises: "Is there an algorithm to decide whether any given instance of PCP has a solution or not?"
10. In other words, "Is PCP decidable?"

PCP is undecidable

Theorem 24.2.1. *PCP is not decidable. (In other words, given any instance of PCP, no algorithm exists to determine whether that instance can be solved.)*

Proof. This Theorem is proved by reducing membership in the universal language to deciding PCP:

1. Given an instance (M, x) of the universal language L_u.
2. Compute a set of tiles, such that if M accepts x, it also is a "yes" instance of the PCP, and vice versa.
3. See the text for details; it is pretty.

□

So what?

We can apply this fact to prove that the above two CFG problems are undecidable:

1. Deciding CFG-intersection decides PCP.
   (a) Given **any** PCP instance $(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)$, let $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_n\}$.
   (b) Let $L(A)$ be all strings we can obtain by concatenating some words from A together, ending with a string of tags c_i to indicate the reversed order of the tiles used.
      i. Example: suppose we append a_1, a_3, a_7. Then $a_1 a_3 a_7 c_7 c_3 c_1$ needs to be in $L(A)$.
      ii. The reversed string of c_i indicates which tiles were included.
   (c) $L(A)$ is context free:
      i. $G_A : A \to \varepsilon \mid a_1 A c_1 \mid a_2 A c_2 \mid \cdots \mid a_n A c_n$.
   (d) And similarly we define B, $L(B)$ and G_B.

(e) If $L(A)$ and $L(B)$ have non-empty words in common, then we can solve the instance of PCP:
   i. Suppose that some word in $L(A)$ equals a word in $L(B)$.
   ii. Then we must have used the same set of tiles (because they both end with the same string of c_is).
   iii. We also created the same word before the c_is.

(f) If $L(A)$ and $L(B)$ have no non-empty words in common, then we cannot solve the instance of PCP.

(g) So if we could decide CFG-intersection, then we could decide PCP, which is undecidable.

(h) Therefore CFG-intersection is undecidable.

2. Deciding CFG-ambiguity decides PCP.
   (a) Consider the grammars G_A for $L(A)$ and G_B for $L(B)$.
   (b) Construct a new grammar G_{AB} with
      i. variables A, B and S (with S as the start variable),
      ii. productions $S \to A|B$,
      iii. all the productions from G_A and
      iv. all the productions from G_B.
   (c) With this construction completed, we now have the desired result by the following Theorem.
   (d) **Theorem:** G_{AB} is ambiguous if and only if the instance (A, B) of PCP has a solution.
   (e) **Proof:** ("If")
      i. Suppose that the indices $i_1, i_2, \ldots, i_m$ are a solution to this instance of PCP.
      ii. Then we have these derivations in G_{AB}:

$$S \Rightarrow A \Rightarrow a_{i_1} A c_{i_1} \Rightarrow a_{i_1} a_{i_2} A c_{i_2} c_{i_1} \Rightarrow \cdots \Rightarrow a_{i_1} \cdots a_{i_m} c_{i_m} \cdots c_{i_1}$$

$$S \Rightarrow B \Rightarrow b_{i_1} B c_{i_1} \Rightarrow b_{i_1} b_{i_2} B c_{i_2} c_{i_1} \Rightarrow \cdots \Rightarrow b_{i_1} \cdots b_{i_m} c_{i_m} \cdots c_{i_1}$$

      iii. By assumption, we have that $a_{i_1} \cdots a_{i_m} = b_{i_1} \cdots b_{i_m}$, i.e both derivations yield the same terminal string.
      iv. Since the derivations are distinct by construction, therefore we conclude that G_{AB} is ambiguous.
   (f) ("Only if")
      i. Assume that G_{AB} is ambiguous.
      ii. The grammars G_A and G_B are unambiguous, because of the trailing tile markers.

iii. So the only way that a terminal string can have two different derivations in G_{AB} is if one derivation starts with $S \to A$ and the other starts with $S \to B$.
iv. The string with two different derivations has a tail $c_{i_m} \cdots c_{i_1}$, for some $m \geq 1$.
v. This tail gives a solution to the instance of PCP, because what precedes the tail is $a_{i_1} \cdots a_{i_m}$ in the first derivation and $b_{i_1} \cdots b_{i_m}$ in the second, and by assumption these must be equal.

24.3 Course Wrap-up

What questions do you have?

24.4 Course Evaluations

`https://perceptions.uwaterloo.ca/`