



Discrete II

Theory of Computation

Charles E. Hughes

COT 4210 – Fall 2019 Notes

Who, What, Where and When

- **Instructor:** Charles Hughes;
Harris Engineering 247C;
e-mail: charles.e.hughes@knights.ucf.edu
(e-mail is a good way to get me)
Please use Subject: COT4210
- **Web Page:** <http://www.cs.ucf.edu/courses/cot4210/Fall2019>
- **Meetings:** TR 4:30PM – 5:45PM, BA1-119;
29-30 class periods, each 75 minutes long.
Office Hours: TR 2:00PM – 3:30PM in HEC-247C
- **TA1:** Stephen Powell
e-mail: stephenmpowell@knights.ucf.edu
Please use Subject: COT4210
Office Hours: MW1500-1630 (3:00PM-4:30PM) in HEC-308
- **TA2:** Trevor Bland
e-mail: tbland96@knights.ucf.edu
Please use Subject: COT4210
Office Hours: F1000-1200 (10:00AM-12:00PM) in HEC-308

Text Material

- This and other material linked from web site.
- Text:
 - **Sipser, *Introduction to the Theory of Computation 2nd or 3rd Ed.*, Course Technologies, 2005/2013.**
 - **Focus on Chapters 1-5,7**
- Reference:
 - **Hopcroft, Motwani and Ullman, *Introduction to Automata Theory, Languages and Computation 3rd Ed.*, Addison-Wesley, 2006.**

Expectations

- **Prerequisites:** COT3100 (discrete structure I); COP3503 (undergraduate algorithm design and analysis).
- **Assignments:** Assignments (likely 10 of them) will be graded but there will also be ungraded practice problems.
- **Exams:** One midterm and a final.
- **Quizzes:** I don't plan on them, but I'll keep that option open.
- **Material:** I will draw heavily from the text by Sipser (Chapters 1-5 and 7). Some material will also come from Hopcroft. Class notes and in-class discussions are, however, comprehensive and cover models, closure properties and undecidable problems that may not be addressed in either of these texts. Note, however, that the Notes are often guidelines to topics in the text, so do not ignore Sipser.

Goals of Course

- Introduce Theory of Computation, including
 - Various models of computation
 - Finite-State Automata and their relation to regular expressions, regular equations and regular grammars
 - Push Down Automata and their relation to context-free languages
 - Techniques for showing languages are NOT in particular language classes
 - Closure and non-closure problems
 - Limits of computation
 - Turing Machines and other equivalent models
 - Decision problems; Undecidable decision problems
 - The technique of reducibility
 - The ubiquity of undecidability (Rice's Theorem)
 - Complexity theory
 - Order notation (this should be a review)
 - Time complexity, the sets P, NP, NP-Hard, NP-Complete and the question does $P=NP$?
 - Reducibility in context of complexity

Expected Outcomes

- You will gain a solid understanding of various types of automata and other computational models and their relation to formal languages.
- You will have a strong sense of the limits that are imposed by the very nature of computation, and the ubiquity of unsolvable problems throughout CS.
- You will understand the notion of computational complexity and especially of the classes of problems known as P, NP, NP-Hard and NP-complete.
- You will come away with stronger formal proof skills and a better appreciation (I hope) of the importance of discrete mathematics to all aspects of CS.

Keeping Up

- I expect you to visit the course web site regularly (preferably daily) to see if changes have been made or material has been added.
- Attendance is preferred, although I do not take roll. I can say that a class where the culture is to come to class does better than one where skipping class is the norm.
- I do, however, ask questions in class and give many hints about the kinds of questions I will ask on exams. It would be a shame to miss the hints, or to fail to impress me with your insightful in-class answers.
- You are responsible for all material covered in class, whether in the text or not.

Rules to Abide By

- Do Your Own Work
 - When you turn in an assignment, you are implicitly telling me that these are the fruits of your labor. Do not copy anyone else's homework or let anyone else copy yours. In contrast, working together to understand lecture material and solutions to problems not posed as graded assignments is encouraged.
- Late Assignments
 - I will accept no late assignments, except under very unusual conditions, and those exceptions must be arranged with me or the GTA in advance unless associated with some tragic event.
- Exams
 - No communication during exams, except with me or a designated proctor, will be tolerated. A single offense will lead to termination of your participation in the class, and the assignment of a failing grade.

Important Dates

- Midterm – Tentatively Thursday, October 17
- Withdraw Deadline – Friday, November 1
- Final – Thursday, Dec. 5, 4:00PM–6:50PM
- Known Days Off: 8/29 (FAMU Game), 11/28 (Thanksgiving)
- Midterm exam date is subject to change with appropriate notice. Final exam is, of course, fixed in stone.

Evaluation (tentative)

- Mid Term – 150 points
- Final Exam – 200 points
- Assignments – 75 points
- Bonus – better exam (midterm or final) weighed +75 points (weight change, not free points)
- Total Available: 500
- Grading will be $A \geq 90\%$, $A- \geq 88\%$,
 $B+ \geq 85\%$, $B \geq 80\%$, $B- \geq 78\%$,
 $C+ \geq 75\%$, $C \geq 70\%$, $C- \geq 60\%$,
 $D \geq 50\%$, $F < 50\%$

Navigating Notes

- When a slide is presenting a problem set, I will highlight the slide title in **Red**
- When a topic is not in the text, I will highlight the slide title in **Green**
- When a topic is covered either in part or only in exercises in the text, I will highlight the slide title in **Blue**
- Oh, and I will occasionally mess up and get the color wrong.

Assignment # 1 Includes Financial Aid Related Activity

Complete questionnaire (in quizzes category) on Webcourses.

Complete all questions on time for a free five points out of total points for all assignments.

Complete and submit by one minute before Midnight Friday, 8/30.

Preliminaries

Mostly from Sipser Chapter 0

This is review material and is discussed in
<http://www.cs.ucf.edu/courses/cot4210/Fall2019/Notes/COT4210NotesPreliminaries.pdf>

Sets and Sequences

- *Sets* are unordered collections of distinct objects. The size of a set is called its *cardinality*. Sets can have finite, countably infinite or uncountably infinite cardinalities.
- The *empty set* is denoted, \emptyset , and is the set with no members; that is, $\emptyset = \{ \}$. The cardinality of \emptyset is 0.
- *Multisets* or *Bags* are unordered collections of objects where we keep track of repeated elements (usually with a count per element)
- The *cross (Cartesian) product of two sets A and B* is $A \times B = \{ (a,b) \mid a \in A \text{ and } b \in B \}$. Note: (a,b) is a sequence
- While sets have no order, *sequences* have order. We can talk about the *k-th element* of a sequence, but not of a set or multiset. Finite sequences of length k are often called *k-tuples*. A 2-tuple is also called a *pair*.
- Subsets of $A \times B$ define binary relations (mappings) from A into B. Such relations, when many-one, can be *partial* or *total* functions (every element of A has a unique mapping to an element of B). Functions mapping to $\{0,1\}$ or $\{\text{false},\text{true}\}$ are *predicates*.
- Relations over $A \times A$ can be *reflexive*, *symmetric*, and/or *transitive* (together, these define an *equivalence* relation that *partitions A* into disjoint subsets).

Alphabets and Strings

- DEFINITION 1. An *alphabet* Σ is a finite, non-empty set of abstract symbols.
- DEFINITION 2. Σ^* , the set of all strings over the alphabet, S, is given inductively as follows.
 - Basis: $\lambda \in \Sigma^*$ (the *null string* is denoted by λ , it is the string of length 0, that is $|\lambda| = 0$) [text uses ε but I avoid that as hate saying $\varepsilon \in A$; it's really confusing when manually written]
 $\forall a \in \Sigma, a \in \Sigma^*$ (the members of S are strings of length 1, $|a| = 1$)
 - Induction rule: If $x \in \Sigma^*$, and $a \in \Sigma$, then $a \cdot x \in \Sigma^*$ and $x \cdot a \in \Sigma^*$. Furthermore, $\lambda \cdot x = x \cdot \lambda = x$, and $|a \cdot x| = |x \cdot a| = 1 + |x|$.
 - NOTE: “ $a \cdot x$ ” denotes “*a concatenated to x*” and is formed by appending the symbol a to the left end of x . Similarly, $x \cdot a$, denotes appending a to the right end of x . In either case, if x is the null string (λ), then the resultant string is “ a ”.
 - We could have skipped saying $\forall a \in \Sigma, a \in \Sigma^*$, as this is covered by the induction step.

UNIVERSE OF DISCOURSE

USUALLY STRINGS OR NATURAL NUMBERS

DECISION PROBLEMS

S

**Subset of interest,
maybe with ordered
elements**

**For some element,
x, is x in S?**

**Question: How many
subsets of Natural
Numbers are there?
How many languages are
there over some finite
alphabet?**

Example 1: S is set of Primes and x is a natural number; is x in S (is x a prime)?

Example 2: S is an undirected graph (pairs for neighbors); is S 3-colorable?

Example 3: S is a program in C; is S syntactically correct?

Example 4: S is program in C; does S halt on all input?

Example 5: S is a set of strings; is the language S Regular, Context-Free, ... ?

Languages

- DEFINITION 3. Let Σ be an alphabet. A *language over Σ* is a subset, L , of Σ^* .
- Example. Languages over the alphabet $\Sigma = \{a, b\}$.
 - \emptyset (the empty set) is a language over Σ
 - Σ^* (the universal set) is a language over Σ
 - $\{a, bb, aba\}$ (a finite subset of Σ^*) is a language over Σ .
 - $\{ab^n a^m \mid n = m^2, n, m \geq 0\}$ (infinite subset) is a language over Σ .
- DEFINITION 4. Let L and M be two languages over Σ . Then the *concatenation of L with M* , denoted $L \cdot M$ is the set,
 $L \cdot M = \{x \cdot y \mid x \in L \text{ and } y \in M\}$
The concatenation of arbitrary strings x and y is defined inductively as follows.
Basis: When $|x| \leq 1$ or $|y| \leq 1$, then $x \cdot y$ is defined as in Definition 2.
Inductive rule: when $|x| > 1$ and $|y| > 1$, then $x = x' \cdot a$ for some $a \in \Sigma$ and $x' \in \Sigma^*$, where $|x'| = |x| - 1$. Then $x \cdot y = x' \cdot (a \cdot y)$.

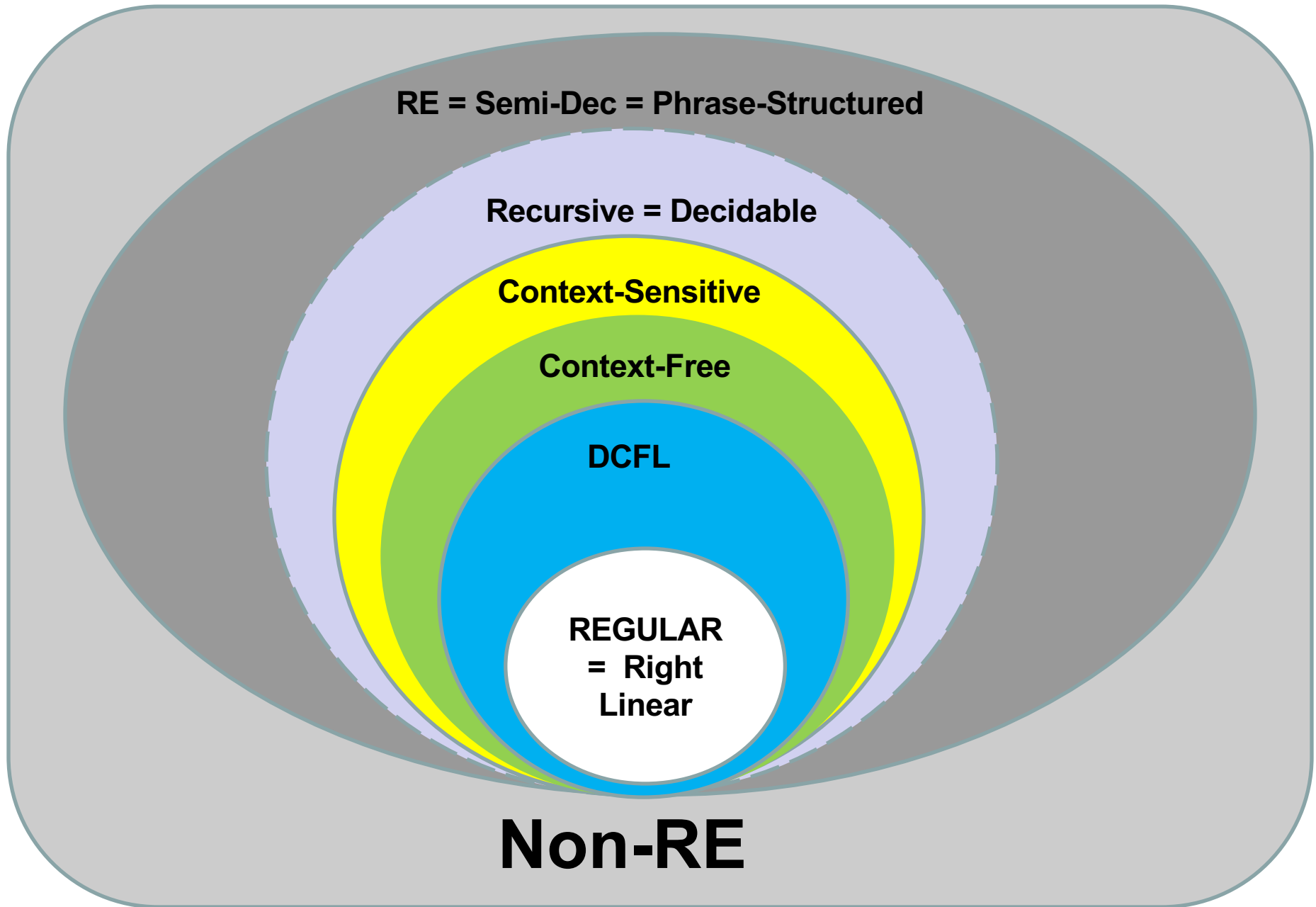
Operations on Strings

- Let s, t be arbitrary strings over Σ
 - $s = a_1 a_2 \dots a_j$, $j \geq 0$, where each $a_i \in \Sigma$
 - $t = b_1 b_2 \dots b_k$, $k \geq 0$, where each $b_i \in \Sigma$
- length: $|s| = j$; $|t| = k$
- concatenate: $= s \cdot t = st = a_1 a_2 \dots a_j b_1 b_2 \dots b_k$; $|st| = j+k$
- power: $s^n = ss \dots s$ (n times) Note: $s^0 = \lambda$
- reverse: $s^R = a_j a_{j-1} \dots a_1$
- substring: for $s = a_1 a_2 \dots a_j$, any $a_p a_{p+1} \dots a_q$ where $1 \leq p \leq q \leq j$ or λ

Properties of Languages

- Let L , M and N be languages over Σ , then:
 - $\emptyset \cdot L = L \cdot \emptyset = \emptyset$
 - $\{\lambda\} \cdot L = L \cdot \{\lambda\} = L$
 - $L \cdot (M \cup N) = L \cdot M \cup L \cdot N$ and $(M \cup N) \cdot L = M \cdot L \cup N \cdot L$
 - Concatenation does **NOT** distribute over **intersection**.
 - $L^0 = \{\lambda\}$ (definition)
 - $L^{n+1} = LL^n = L^nL$, $n \geq 0$. (definition)
 - $L^+ = L^1 \cup L^2 \cup \dots L^n \dots$ (definition)
 - $L^* = L^0 \cup L^1 \cup L^2 \cup \dots L^n \dots$ (definition) = $L^0 \cup L^+$
 - $(L^*)^* = L^*$
 - $(LM)^*L = L(ML)^*$
 - $(L^* \cdot M^*)^* = (L^* \cup M^*)^* = (L \cup M)^*$
 - $(L^0 \cup L^1 \cup L^2 \cup \dots L^n)L^* = L^*$, for all $n \geq 0$.

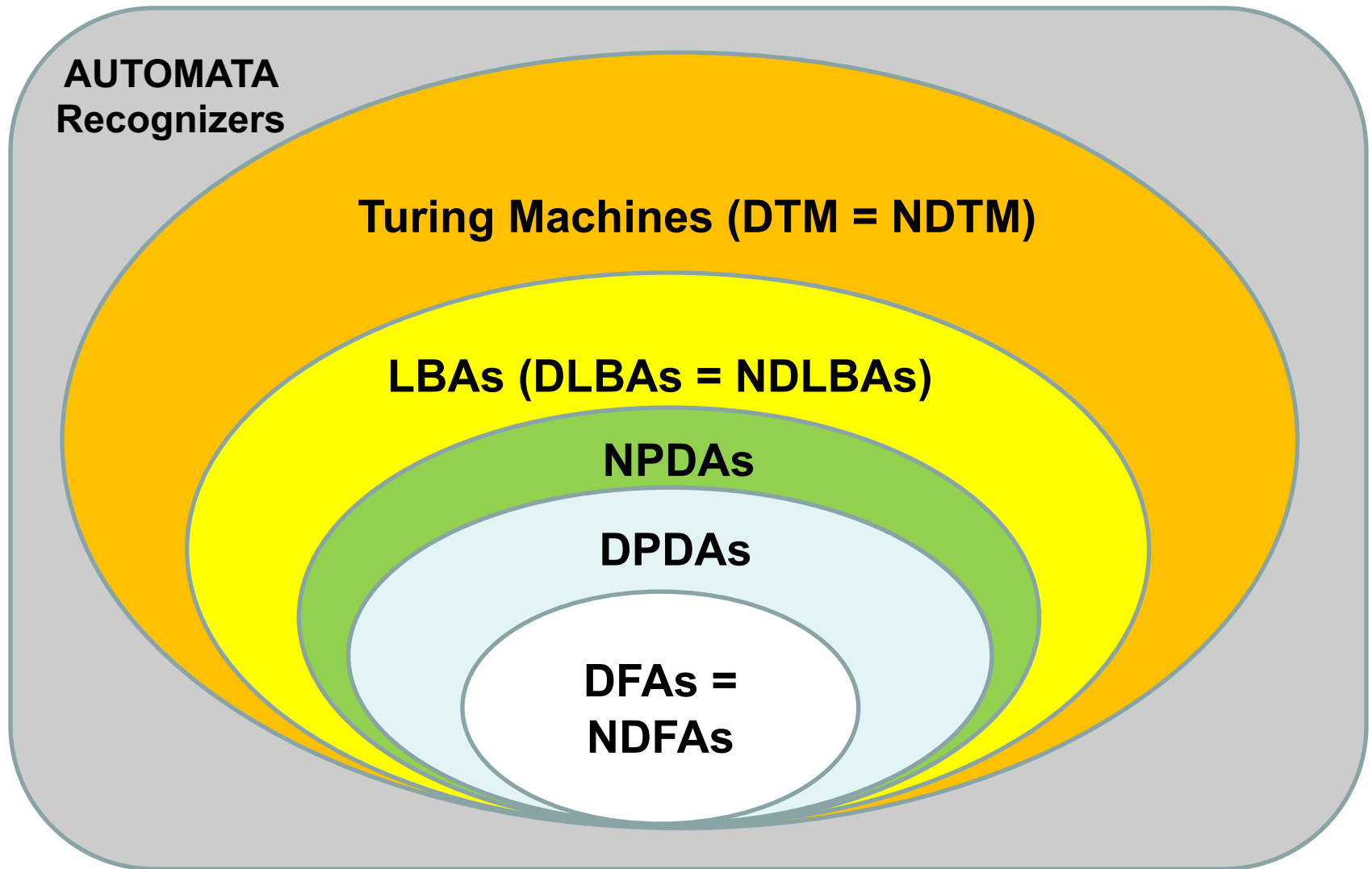
UNIVERSE OF LANGUAGES



Recognizers and Generators

1. When we discuss languages and classes of languages, we discuss recognizers and generators
2. A recognizer for a specific language is a program or computational model that differentiates members from non-members of the given language
3. A portion of the job of a compiler is to check to see if an input is a legitimate member of some specific programming language – we refer to this as a syntactic recognizer
4. A generator for a specific language is a program that generates all and only members of the given language
5. In general, it is not individual languages that interest us, but rather classes of languages that are definable by some specific class of recognizers or generators
6. One type of recognizer is called an automata and there are multiple classes of automata
7. One type of generator is called a grammar and there are multiple classes of grammars
8. Our first journey will be through automata and grammars

MODELS OF COMPUTATION



Of these models, only TMs can do general computation

REWRITING SYSTEMS

GRAMMARS
Generators

Type 0=Phrase-Structured

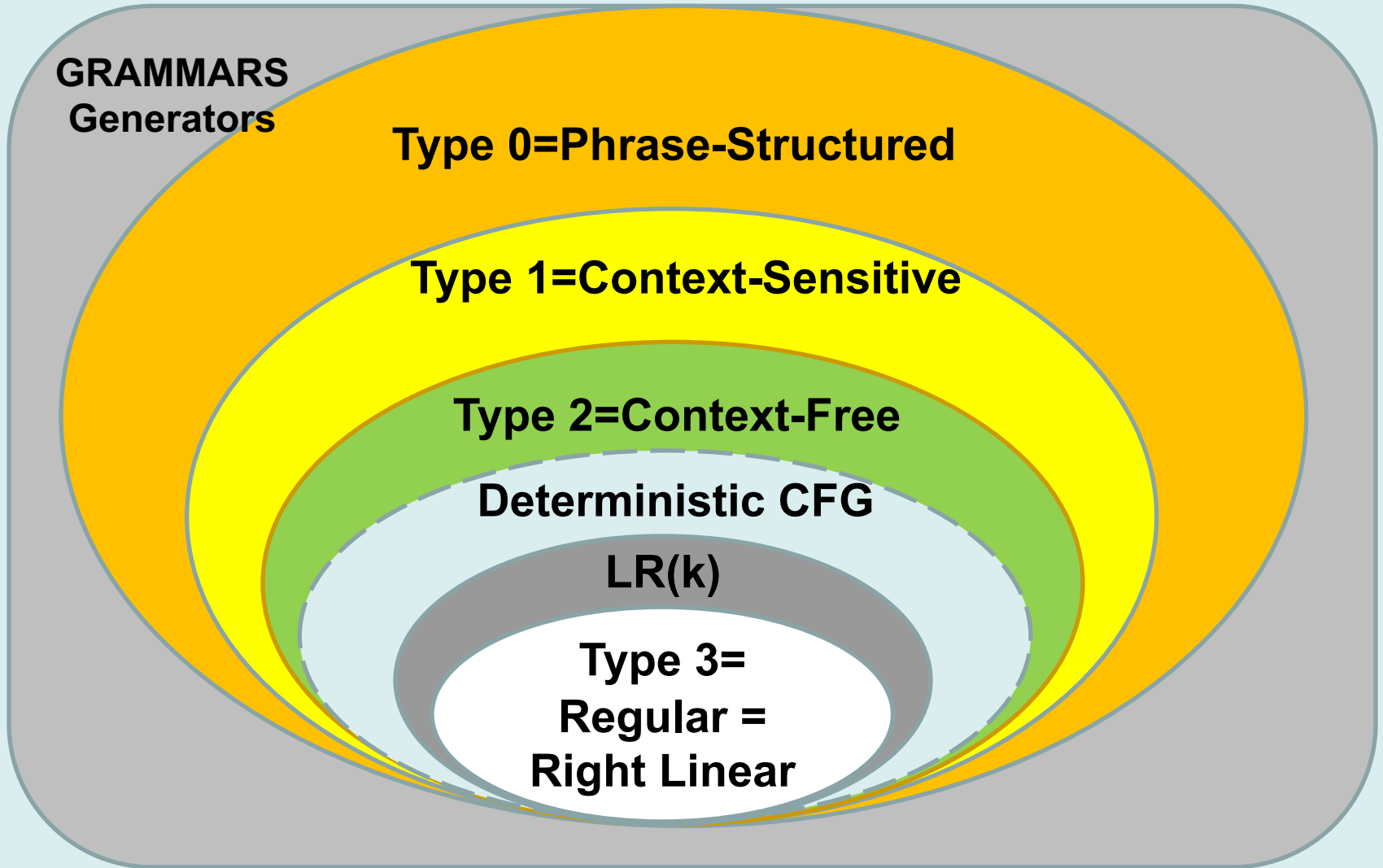
Type 1=Context-Sensitive

Type 2=Context-Free

Deterministic CFG

LR(k)

Type 3=
Regular =
Right Linear



Regular Languages

Includes and Expands on
Chapter 1 of Sipser

Finite-State Automata

- A Finite-State Automaton (FSA) has only one form of memory, its current state.
- As any automaton has a predetermined finite number of states, this class of machines is quite limited, but still very useful.
- There are two classes: Deterministic Finite-State Automata (DFAs) and Non-Deterministic Finite-State Automata (NFAs)
- We focus on DFAs for now.

Concrete Model of FSA

L is a finite-state (regular) language over finite alphabet Σ

Each x_i is a character in Σ

$w = x_1 x_2 \dots x_n$ is a string to be tested for membership in L



q_0 

- Arrow above represents read head that starts on left.
- $q_0 \in Q$ (finite state set) is initial state of machine.
- Only action at each step is to change state based on character being read and current state. State change is determined by a transition function $\delta: Q \times \Sigma \rightarrow Q$.
- Once state is changed, read head moves right.
- Machine stops when head passes last input character.
- Machine accepts a string as a member of L if it ends up in a state from Final State set $F \subseteq Q$.

Deterministic Finite-State Automata (DFA)

- A deterministic finite-state automaton (DFA) A is defined by a 5-tuple $A = (Q, \Sigma, \delta, q_0, F)$, where
 - Q is a finite set of symbols called the states of A
 - Σ is a finite set of symbols called the alphabet of A
 - δ is a function from $Q \times \Sigma$ into Q ($\delta: Q \times \Sigma \rightarrow Q$) called the transition function of A
 - $q_0 \in Q$ is a unique element of Q called the start state
 - F is a subset of Q ($F \subseteq Q$) called the final states (can be empty)


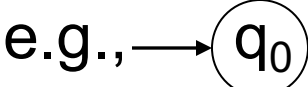
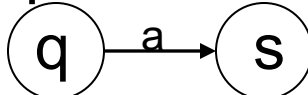

DFA Transitions

- Given a DFA, $A = (Q, \Sigma, \delta, q_0, F)$, we can define the reflexive transitive closure of δ , $\delta^*: Q \times \Sigma^* \rightarrow Q$, by
 - $\delta^*(q, \lambda) = q$ where λ is the string of length 0
 - Note that text uses ϵ rather than λ as symbol for string of length zero
 - $\delta^*(q, ax) = \delta^*(\delta(q, a), x)$, where $a \in \Sigma$ and $x \in \Sigma^*$
 - Note that this means $\delta^*(q, a) = \delta(q, a)$, where $a \in \Sigma$ as $a = a\lambda$
- We also define the transitive closure of δ , δ^+ , by
 - $\delta^+(q, w) = \delta^*(q, w)$ when $|w| > 0$ or, equivalently, $w \in \Sigma^+$
- The function δ^* describes every step of computation by the automaton starting in some state until it runs out of characters to read

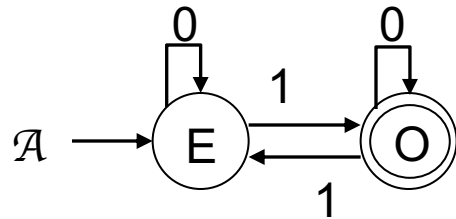
Regular Languages and DFAs

- Given a DFA, $A = (Q, \Sigma, \delta, q_0, F)$, we can define the language accepted by A as those strings that cause it to end up in a final state once it has consumed the entire string
- Formally, the language accepted by A is
 - $\{ w \mid \delta^*(q_0, w) \in F \}$
- We generally refer to this language as $L(A)$
- We define the notion of a Regular Language by saying that a language is Regular if and only if it is accepted (recognized) by some DFA

State Diagram

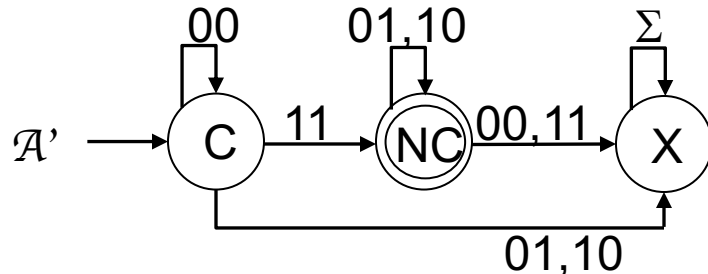
- A finite-state automaton can be described by a state diagram, where
 - Each state is represented by a node labelled with that state, e.g., 
 - The start state has an arc entering it with no source, e.g., 
 - Each transition $\delta(q,a) = s$ is represented by a directed arc from node q to node s that is labelled with the letter a , e.g., 
 - Each final state has an extra circle around its node, e.g., 

Sample DFAs # 1, 2



$\mathcal{A} = (\{E,O\}, \{0,1\}, \delta, E, \{O\})$, where δ is defined by above diagram.

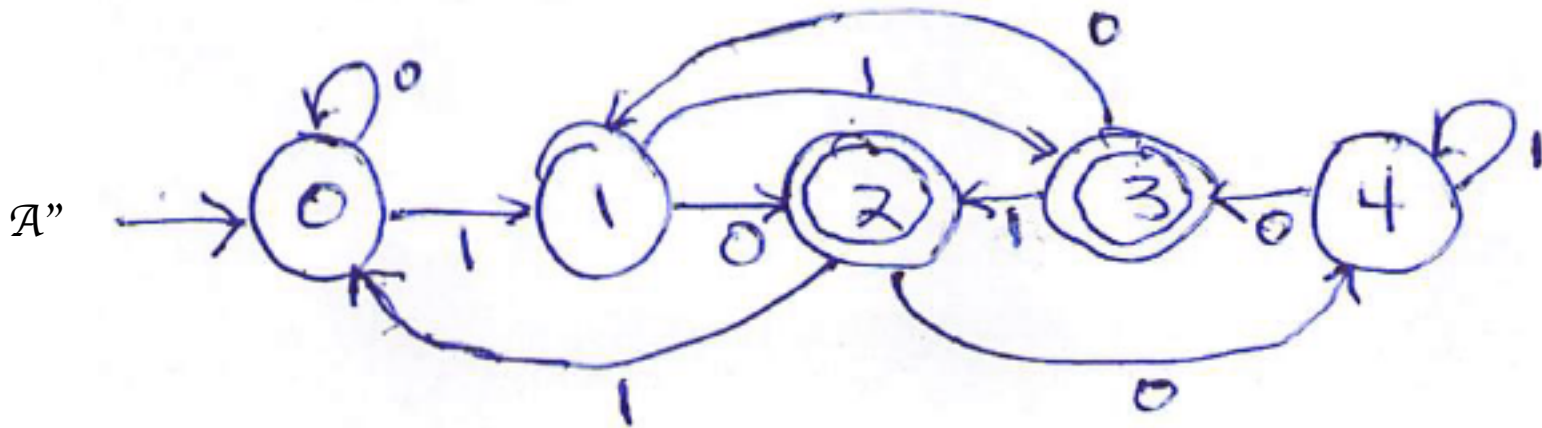
$L(\mathcal{A}) = \{ w \mid w \text{ is a binary string of odd parity} \}$



$\mathcal{A}' = (\{C,NC,X\}, \{00,01,10,11\}, \delta', C, \{NC\})$, where δ' is defined by above diagram.

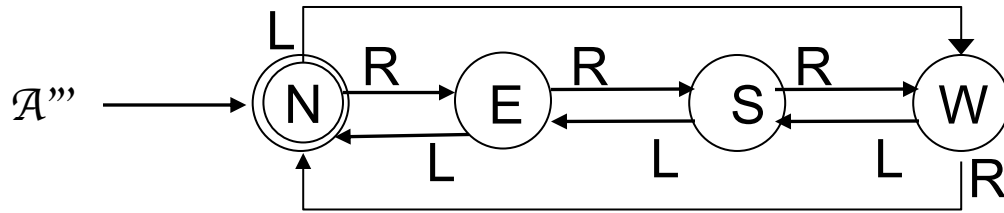
$L(\mathcal{A}') = \{ w \mid w \text{ is a pair of binary strings where the bottom string is the 2's complement of the top one, both read least (lsb) to most significant bit (msb)} \}$

Sample DFA # 3



$\mathcal{A}'' = (\{0,1,2,3,4\}, \{0,1\}, \delta'', 0, \{2,3\})$, where δ'' is defined by above diagram. $L(\mathcal{A}'') = \{w \mid w \text{ is a binary string that, read left to right (msb to lsb), when interpreted as a decimal number divided by 5, has a remainder of 2 or 3}\}$

Sample DFA # 4



$\mathcal{A}''' = (\{N,E,W,S\}, \{R,L\}, \delta''', N, \{N\})$, where δ''' is defined by above diagram.
 $L(\mathcal{A}''')$ = { w | w is a set of commands passed to a sentinel that starts facing North and changes directions R(ight)/clockwise or L(eft)/counterclockwise based on the corresponding input character. w must eventually lead the sentinel back to facing North }

Assignment # 2

Write a deterministic finite-state automaton (DFA) that describes the operations of a sentinel that accepts only the commands R and L. The sentinel can look North, East, South or West and always starts looking North. R causes the sentinel to turn clockwise 90 degrees. L causes the sentinel to turn counter-clockwise 90 degrees. A sequence of actions is acceptable if the sentinel has looked, in addition to North, to East, South and West, in any order that is possible.

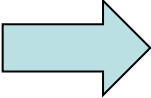
Hint: States should look like $\langle \text{DirectionsVisited}, \text{CurrentDirection} \rangle$. For example, the DFA starts in $\langle \{N\}, N \rangle$. Let's say it received commands RLL, then it would be in state $\langle \{N, E, W\}, W \rangle$. If it gets another L command, then it enters a final state and stays there no matter what additional commands are received. This can easily be done in 15 states, but it can be done in fewer as well. No optimization is required, except to not exceed 15 states.

Due Tuesday, Sept. 10 at 11:59PM (use Webcourses to turn in)

State Transition Table

- A finite-state automaton can be described by a state transition table with $|Q|$ rows and $|\Sigma|$ columns
- Rows are labelled with state names and columns with input letters
- The start state has some indicator, e.g., a greater than sign ($>q$) and each final state has some indicator, e.g., an underscore (\underline{f})
- The entry in row q , column a , contains $\delta(q,a)$
- In general we will use state diagrams, but transition tables are useful in some cases (state minimization)

Sample DFA # 4

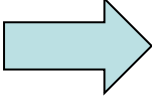
		0	1
	0 % 5	0 % 5	1 % 5
	1 % 5	2 % 5	3 % 5
	2 % 5	4 % 5	0 % 5
Accept State	<u>3 % 5</u>	1 % 5	2 % 5
	4 % 5	3 % 5	4 % 5

$\mathcal{A}''' = (\{0\%5, 1\%5, 2\%5, 3\%5, 4\%5\}, \{0, 1\}, \delta''', 0, \{3\%5\})$, where δ''' is defined by above diagram.

$L(\mathcal{A}''') = \{ w \mid w \text{ is a binary string of length at least 1 being read left to right (msb to lsb) that, when interpreted as a decimal number divided by 5, has a remainder of 3 } \}$

Really, this is better done as a state diagram similar to what you saw earlier but have put this up so you can see the pattern.

Sample DFA # 5



	A-Z	a-z	0-9	@#\$\$%^&
⇒ Empty	A	a	0	@
A	A	Aa	A0	A@
a	Aa	a	a0	a@
0	A0	a0	0	0@
@	A@	a@	0@	@
Aa	Aa	Aa	Aa0	Aa@
A0	A0	Aa0	A0	A0@
A@	A@	Aa@	A0@	A@
a0	Aa0	a0	a0	a0@
a@	Aa@	a@	a0@	a@
0@	A0@	a0@	0@	0@
Aa0	Aa0	Aa0	Aa0	Aa0@
Aa@	Aa@	Aa@	Aa0@	Aa@
A0@	A0@	Aa0@	A0@	A0@
a0@	Aa0@	a0@	a0@	a0@
Aa0@	Aa0@	Aa0@	Aa0@	Aa0@

This checks a string to see if it's a legal password. In our case, a legal password must contain at least one of each of the following: lower case letter, upper case letter, number, and special character from the following set {!@#\$%^&}. No other characters are allowed

FSAs and Applications

- A synchronous sequential circuit has
 - Binary input lines (input admitted at clock tick)
 - Binary output lines (simple case is one line)
 - 1 accepts; 0 rejects input
 - Internal flip flops (memory) that define state
 - Simple combinatorial circuits (and, or, not) that combine current state and input to alter state
 - Simple combinatorial circuits (and, or, not) that use state to determine output
- Think about FSA to recognize the string PAPANAT appearing somewhere in a corpus of text, say with a substring PAPANATRICK
- Comments about GREP and Lexical Analysis

DFA Closure

- Regular languages (those recognized by DFAs) are closed under complement, union, intersection, difference and exclusive or (\oplus) and many other set operations
- Let $A_1 = (Q_1, \Sigma, \delta_1, q_0, F_1)$, $A_2 = (Q_2, \Sigma, \delta_2, s_0, F_2)$ be arbitrary DFAs
- $\Sigma^*-L(A_1)$ is recognized by $A_1^C = (Q_1, \Sigma, \delta_1, q_0, Q_1 - F_1)$
- Define $A_3 = (Q_1 \times Q_2, \Sigma, \delta_3, \langle q_0, s_0 \rangle, F_3)$ where $\delta_3(\langle q, s \rangle, a) = \langle \delta_1(q, a), \delta_2(s, a) \rangle$, $q \in Q_1$, $s \in Q_2$, $a \in \Sigma$
 - $L(A_1) \cup L(A_2)$ is recognized when $F_3 = (F_1 \times Q_2) \cup (Q_1 \times F_2)$
 - $L(A_1) \cap L(A_2)$ is recognized when $F_3 = F_1 \times F_2$
 - $L(A_1) - L(A_2)$ is recognized when $F_3 = F_1 \times (Q_2 - F_2)$
 - $L(A_1) \oplus L(A_2)$ is recognized when $F_3 = F_1 \times (Q_2 - F_2) \cup (Q_1 - F_1) \times F_2$

Complement of Regular Sets

- Let $A = (Q, \Sigma, \delta, q_0, F)$
- Simply create new automaton
 $A^C = (Q, \Sigma, \delta, q_0, Q-F)$
- $L(A^C) = \{ w \mid \delta^*(q_0, w) \in Q-F \} =$
 $\{ w \mid \delta^*(q_0, w) \notin F \} =$
 $\{ w \mid w \notin L(A) \}$
- When we discuss them shortly, imagine trying to do this in the context of regular expressions
- Choosing the right representation can make a very big difference in how easy or hard it is to prove some property is true

Parallelizing DFAs

- Regular sets can be shown closed under many binary operations using the notion of parallel machine simulation
- Let $A_1 = (Q_1, \Sigma, \delta_1, q_0, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, s_0, F_2)$ where $Q_1 \cap Q_2 = \emptyset$
- $B = (Q_1 \times Q_2, \Sigma, \delta_3, \langle q_0, s_0 \rangle, F_3)$ where $\delta_3(\langle q, s \rangle, a) = \langle \delta_1(q, a), \delta_2(s, a) \rangle$
- Union is $F_3 = F_1 \times Q_2 \cup Q_1 \times F_2$
- Intersection is $F_3 = F_1 \times F_2$
 - Can do by combining union and complement
- Difference is $F_3 = F_1 \times (Q_2 - F_2)$
 - Can do by combining intersection and complement
- Exclusive Or is $F_3 = F_1 \times (Q_2 - F_2) \cup (Q_1 - F_1) \times F_2$

Non-determinism NFA

- A non-deterministic finite-state automaton (NFA) A is defined by a 5-tuple $A = (Q, \Sigma, \delta, q_0, F)$, where
 - Q is a finite set of symbols called the states of A
 - Σ is a finite set of symbols called the alphabet of A
 - δ is a function from $Q \times \Sigma_e$ into $P(Q) = 2^Q$; Note: $\Sigma_e = (\Sigma \cup \{\lambda\})$
($\delta: Q \times \Sigma_e \rightarrow P(Q)$) called the transition function of A ; by definition $q \in \delta(q, \lambda)$
 - $q_0 \in Q$ is a unique element of Q called the start state
 - F is a subset of Q ($F \subseteq Q$) called the final states
 - Note that a state/input (called a discriminant) can lead nowhere new, one place or many places in an NFA; moreover, an NFA can jump between states even without reading any input symbol
 - For simplicity, we often extend the definition of $\delta: Q \times \Sigma_e$ to a variant that handles sets of states, where $\delta: P(Q) \times \Sigma_e$ is defined as $\delta(S, a) = \cup_{q \in S} \delta(q, a)$, where $a \in \Sigma_e$ – if $S = \emptyset$, $\cup_{q \in S} \delta(q, a) = \emptyset$

NFA Transitions

- Given an NFA, $A = (Q, \Sigma, \delta, q_0, F)$, we can define the reflexive transitive closure of δ , $\delta^*: P(Q) \times \Sigma^* \rightarrow P(Q)$, by
 - λ -Closure(S) = $\{ t \mid t \in \delta^*(S, \lambda) \}$, $S \in P(Q)$ – extended δ
 - $\delta^*(S, \lambda) = \lambda$ -Closure(S)
 - $\delta^*(S, ax) = \delta^*(\lambda$ -Closure($\delta(S, a)$), x), where $a \in \Sigma$ and $x \in \Sigma^*$
 - Note that $\delta^*(S, ax) = \bigcup_{q \in S} \bigcup_{p \in \lambda$ -Closure($\delta(q, a)$) $\delta^*(p, x)$, where $a \in \Sigma$ and $x \in \Sigma^*$
- We also define the transitive closure of δ , δ^+ , by
 - $\delta^+(S, w) = \delta^*(S, w)$ when $|w| > 0$ or, equivalently, $w \in \Sigma^+$
- The function δ^* describes every “possible” step of computation by the non-deterministic automaton starting in some state until it runs out of characters to read

NFA Languages

- Given an NFA, $A = (Q, \Sigma, \delta, q_0, F)$, we can define the language accepted by A as those strings that allow it to end up in a final state once it has consumed the entire string – here we just mean that there is some accepting path
- Formally, the language accepted by A is
 - $\{ w \mid (\delta^*(\lambda\text{-Closure}(\{q_0\}), w) \cap F) \neq \emptyset \}$
- Notice that we accept if there is any set of choices of transitions that lead to a final state

Finite-State Diagram

- A non-deterministic finite-state automaton can be described by a finite-state diagram, except
 - We now can have transitions labelled with λ
 - The same letter can appear on multiple arcs from a state q to multiple distinct destination states

Equivalence of DFA and NFA

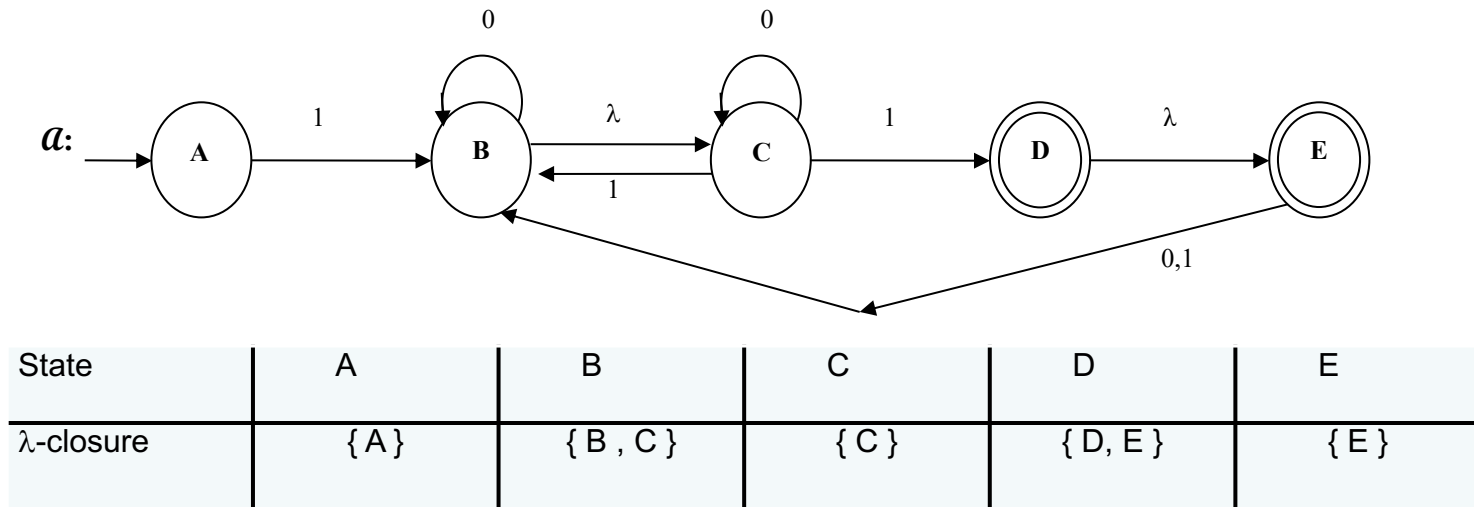
- Clearly every DFA is an NFA except that $\delta(q,a) = s$ becomes $\delta(q,a) = \{s\}$, so any language accepted by a DFA can be accepted by an NFA.
- The challenge is to show every language accepted by an NFA is accepted by an equivalent DFA. That is, if A is an NFA, then we can construct a DFA A' , such that $L(A') = L(A)$.

Constructing DFA from NFA

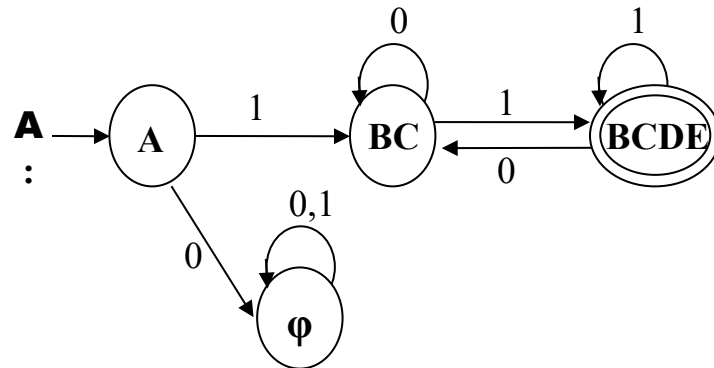
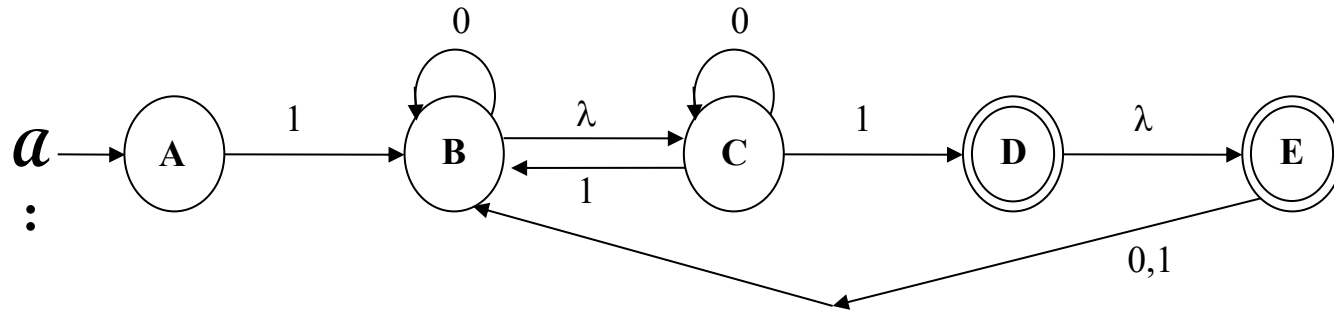
- Let $A = (Q, \Sigma, \delta, q_0, F)$ be an arbitrary NFA
- Let S be an arbitrary subset of Q .
 - Construct the sequence $\text{seq}(S)$ to be a sequence that contains all elements of S in lexicographical order, using angle brackets to . That is, if $S = \{q_1, q_3, q_2\}$ then $\text{seq}(S) = \langle q_1, q_2, q_3 \rangle$. If $S = \emptyset$ then $\text{seq}(S) = \langle \rangle$
- Our goal is to create a DFA, A' , whose state set contains $\text{seq}(S)$, whenever there is some w such that $S = \delta^*(q_0, w)$
- To make our life easier, we will act as if the states of A' are sets, knowing that we really are talking about corresponding sequences

λ -Closure

- Define the λ -Closure of a state q as the set of states one can arrive at from q , without reading any additional input.
- Formally $\lambda\text{-Closure}(q) = \{ t \mid t \in \delta^*(q, \lambda) \}$
- We can extend this to $S \in P(Q)$ by
 $\lambda\text{-Closure}(S) = \{ t \mid t \in \delta^*(q, \lambda), q \in S \} = \{ t \mid t \in \lambda\text{-Closure}(q), q \in S \}$



DFA from NFA



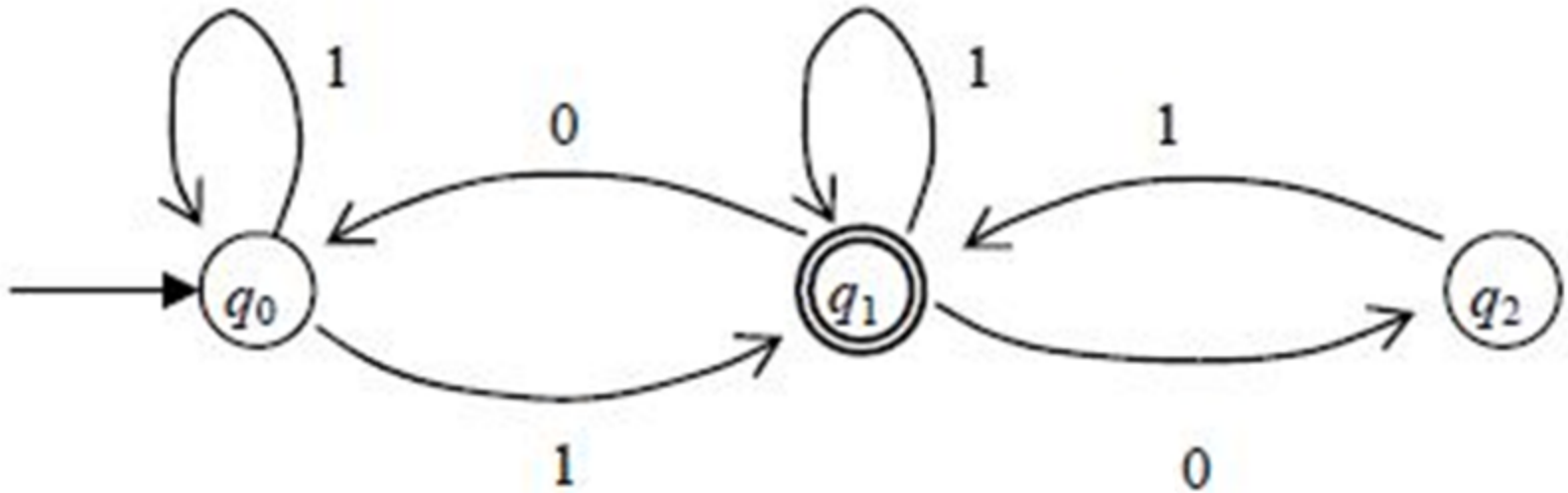
Details of DFA

- Let $A = (Q, \Sigma, \delta, q_0, F)$ be an arbitrary NFA
- In an abstract sense,
 $A' = (\langle P(Q) \rangle, \Sigma, \delta', \langle \lambda\text{-Closure}(\{q_0\}) \rangle, F')$,
where $P(Q)$ is the power set of Q , but we really don't need so many states ($2^{|Q|}$) and we can iteratively determine those needed by starting at $\lambda\text{-Closure}(\{q_0\})$ and keeping only states reachable from here
- Define $\delta'(\langle S \rangle, a) = \langle \lambda\text{-Closure}(\delta(S, a)) \rangle = \langle \bigcup_{q \in S} \lambda\text{-Closure}(\delta(q, a)) \rangle$, where $a \in \Sigma$, $S \in P(Q)$
- $F' = \{ \langle S \rangle \in \langle P(Q) \rangle \mid (S \cap F) \neq \emptyset \}$

Regular Languages and NFAs

- Showing that every DFA can be simulated by an NFA that accepts the same language and every NFA can be simulated by a DFA that accepts the same language proves the following
- A language is Regular if and only if it is accepted (recognized) by some NFA
- We now have two equivalent classes of recognizers for Regular Languages

Simple Exercise: Convert from NFA to DFA



Regular Expressions

Regular Sets

Regular Expressions

- Primitive:
 - Φ denotes $\{\}$
 - λ denotes $\{\lambda\}$
 - a where a is in Σ denotes $\{a\}$
- Closure:
 - If R and S are regular expressions then so are $R \cdot S$, $R + S$ and R^* , where
 - $R \cdot S$ denotes $RS = \{xy \mid x \text{ is in } R \text{ and } y \text{ is in } S\}$
 - $R + S$ denotes $R \cup S = \{x \mid x \text{ is in } R \text{ or } x \text{ is in } S\}$
 - R^* denotes R^*
- Parentheses are used as needed

Lexical Analysis

- Consider distinguishing variable names from keywords like
 - IF `return(IFS);`
 - INT `return(INT);`
 - `[a-zA-Z]([a-zA-Z0-9_])*` `return(IDENT);`
 - Equivalent to `a+b+...+z`, etc.
- This really screams for non-determinism
 - With added constraints of finding longest/first match
- Non-deterministic automata typically have fewer states
- However, non-deterministic FSA (NFA) interpretation is not as fast as deterministic

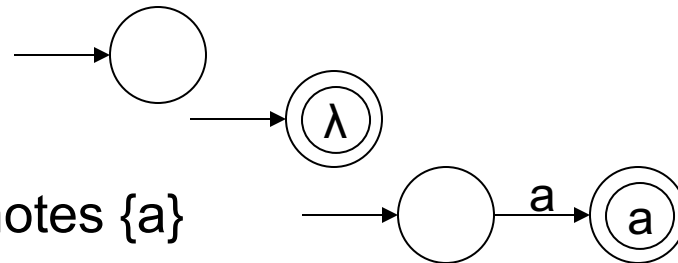
Regular Sets = Regular Languages

- Show every regular expression denotes a language recognized by a finite-state automaton (can do deterministic or non-deterministic)
- Show every Finite-State Automata recognizes a language denoted by a regular expression

Every Regular Set is a Regular Language

- Primitive:

- Φ denotes $\{\}$
- λ denotes $\{\lambda\}$
- a where a is in Σ denotes $\{a\}$



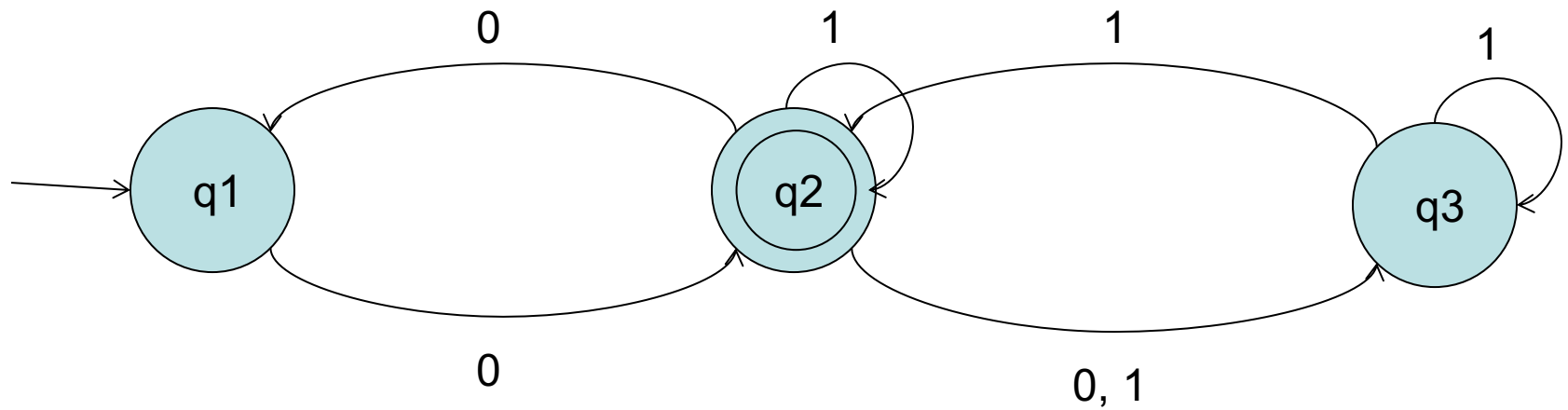
- Closure: (Assume that R's and S's states do not overlap)

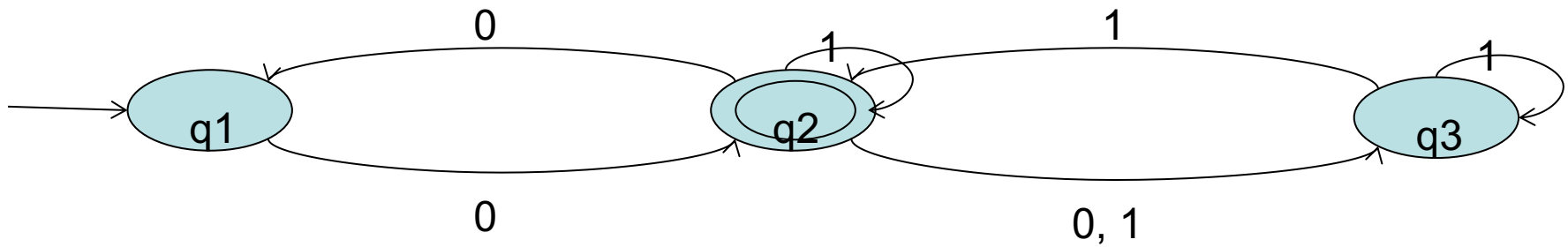
- $R \cdot S$ start with machine for R, add λ transitions from every final state of R's recognizer to start state of S, making final state of S final states of new machine
- $R + S$ create new start state and add λ transitions from new state to start states of each of R and S, making union of R's and S's final states the new final states
- R^* add λ transitions from each final state of R back to its start state, keeping original start and final states (gets R^+) – FIX?

Every Regular Language is a Regular Set Using R_{ij}^k

- This is a challenge that can be addressed in multiple ways but I like to start with the R_{ij}^k approach. Here's how it works.
- Let $A = (Q, \Sigma, \delta, q_1, F)$ be a DFA, where $Q = \{q_1, q_2, \dots, q_n\}$
- $R_{ij}^k = \{w \mid \delta^*(q_i, w) = q_j, \text{ and no intermediate state visited between } q_i \text{ and } q_j, \text{ while reading } w, \text{ has index } > k\}$
- Basis: $k=0$, $R_{ij}^0 = \{a \mid \delta(q_i, a) = q_j\}$ sets are either Φ , λ , or an element of Σ or $\lambda + \text{element of } \Sigma$, and so are regular sets
- Inductive hypothesis: Assume R_{ij}^m are regular sets for $0 \leq m \leq k$
- Inductive step: $k+1$, $R_{ij}^{k+1} = (R_{ij}^k + R_{ik+1}^k \cdot (R_{k+1k+1}^k)^* \cdot R_{k+1j}^k)$
- $L(A) = \bigcup_{f \in F} R_{1f}^n$

Convert to RE





- $R_{11}^0 = \lambda$
- $R_{21}^0 = 0$
- $R_{31}^0 = \phi$
- $R_{11}^1 = \lambda$
- $R_{21}^1 = 0$
- $R_{31}^1 = \phi$
- $R_{11}^2 = \lambda + 0(1+00)^*0$
- $R_{21}^2 = (1+00)^*0$
- $R_{31}^2 = 1(1+00)^*0$
- $L = R_{12}^3 =$
 $0(1+00)^* + 0(1+00)^*(0+1) (1+1(1+00)^*(0+1))^* 1(1+00)^*$
- $R_{12}^0 = 0$
- $R_{22}^0 = \lambda + 1$
- $R_{32}^0 = 1$
- $R_{12}^1 = 0$
- $R_{22}^1 = \lambda + 1 + 00$
- $R_{32}^1 = 1$
- $R_{12}^2 = 0(1+00)^*$
- $R_{22}^2 = (1+00)^*$
- $R_{32}^2 = 1(1+00)^*$
- $R_{13}^0 = \phi$
- $R_{23}^0 = 0 + 1$
- $R_{33}^0 = \lambda + 1$
- $R_{13}^1 = \phi$
- $R_{23}^1 = 0 + 1$
- $R_{33}^1 = \lambda + 1$
- $R_{13}^2 = 0(1+00)^*(0+1)$
- $R_{23}^2 = (1+00)^*(0+1)$
- $R_{33}^2 = \lambda + 1 + 1(1+00)^*(0+1)$

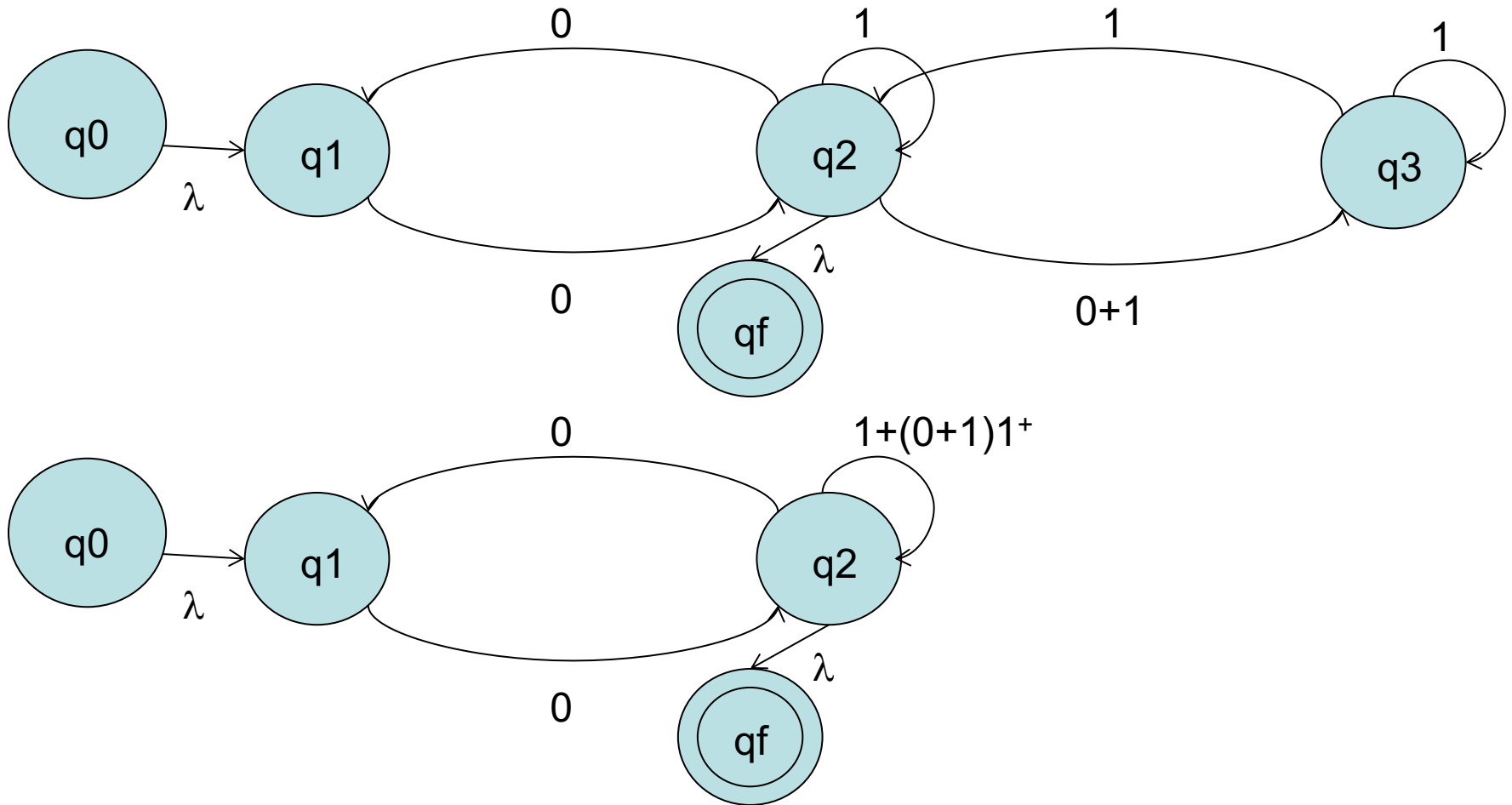
State Ripping Concept

- This is similar to generalized automata approach but with fewer arcs than text. It actually gets some of its motivation from R_{ij}^k approach as well
- Add a new start state and add a λ -transition to existing start state
- Add a new final state q_f and insert λ -transitions from all existing final states to the new one; make the old final states non-final
- Leaving the start and final states, successively pick states to remove
- For each state to be removed, change the arcs of every pair of externally entering and exiting arcs to reflect the regular expression that describes all strings that could result in such a double transition; be sure to account for loops in the state being removed. Also, or (+) together expressions that have the same start and end nodes
- When have just start and final, the regular expression that leads from start to final describes the associated regular set

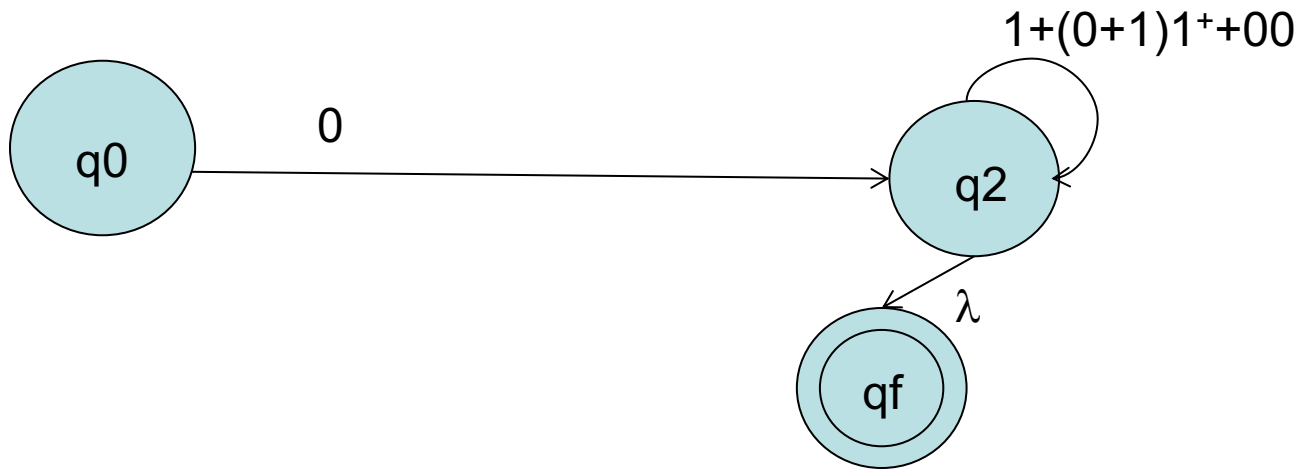
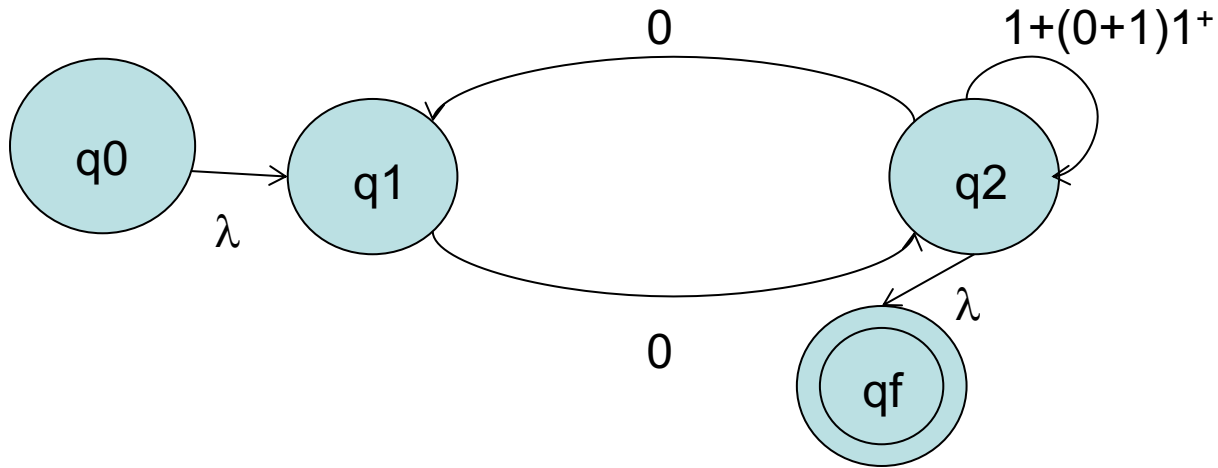
State Ripping Details

- Let B be the node to be removed
- Let e_1 be the regular expression on the arc from some node A to some node B ($A \neq B$); e_2 be the expression from B back to B (or λ if there is no recursive arc); e_3 be the expression on the arc from B to some other node C ($C \neq B$ but C could be A); e_4 be the expression from A to C
- Erase the existing arcs from A to B and A to C, adding a new arc from A to C labelled with the expression $e_4 + e_1 e_2^* e_3$
- Do this for all nodes that have edges to B until B has no more entering edges; at this point remove B and any edges it has to other nodes and itself
- Iterate until all but the start and final nodes remain
- The expression from start to final describes regular set that is equivalent to regular language accepted by original automaton
- Note: Your choices of the order of removal make a big difference in how hard or easy this is

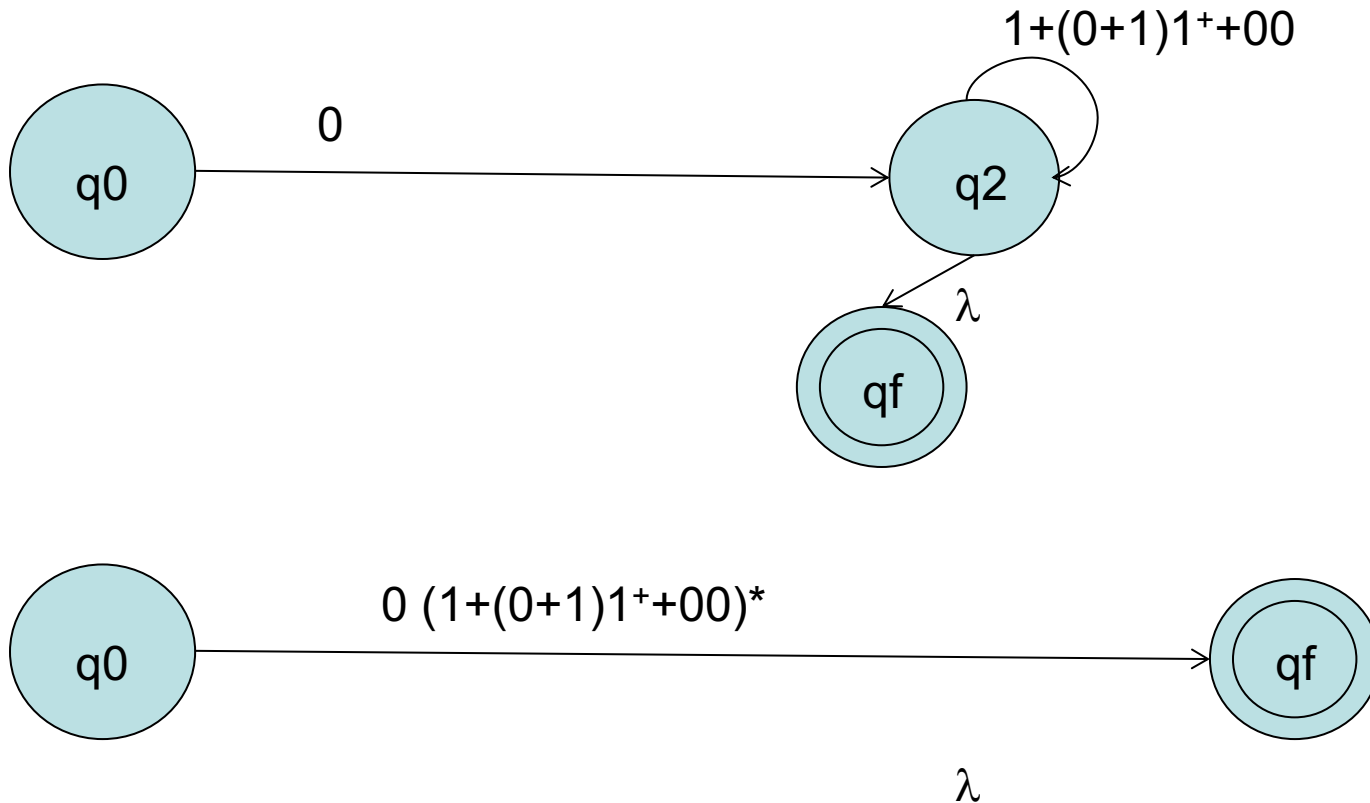
Use Ripping; Rip q3



Use Ripping; Rip q1



Use Ripping; Rip q2



$$L = 0 (1+(0+1)1^{++}00)^*$$

Regular Equations (Arden)

- Assume that R , Q and P are sets such that P does not contain the string of length zero, and R is defined by
- $R = Q + RP$
- We wish to show that
- $R = QP^*$
- This can be found under “Arden’s Theorem”

Show QP^* is a Solution

- We first show that QP^* is contained in R . By definition, $R = Q + RP$.
- To see if QP^* is a solution, we insert it as the value of R in $Q + RP$ and see if the equation balances
- $R = Q + QP^*P = Q(\lambda + P^*P) = QP^*$
- Hence QP^* is a solution, but not necessarily the only solution.

Uniqueness of Solution

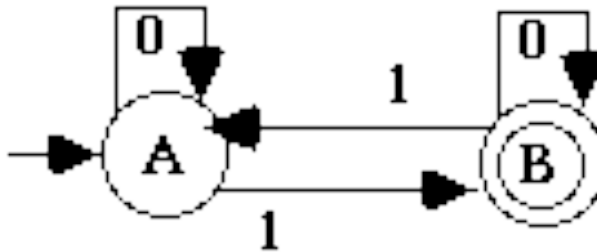
- To prove uniqueness, we show that R is contained in QP^* .
- By definition, $R = Q + RP = Q + (Q + RP)P$
- $= Q + QP + RP^2 = Q + QP + (Q + RP)P^2$
- $= Q + QP + QP^2 + RP^3$
- ...
- $= Q(\lambda + P + P^2 + \dots + P^i) + RP^{i+1}$, for all $i \geq 0$
- Choose any w in R , where $|W| = k$. Then, from above,
- $R = Q(\lambda + P + P^2 + \dots + P^k) + RP^{k+1}$
- but, since P does not contain the string of length zero, w is not in RP^{k+1} . But then w is in
- $Q(\lambda + P + P^2 + \dots + P^k)$ and hence w is in QP^* .

Example

- We use the above to solve simultaneous regular equations. For example, we can associate regular expressions with finite-state automata as follows

- Hence,

- For A, $Q = \lambda + B1$; $P = 0$
 $A = QP^* = (\lambda + B1)0^*$
 $= B10^* + 0^*$



$$A = \lambda + B1 + A0$$

$$B = A1 + B0$$

- $B = B10^*1 + B0 + 0^*1$

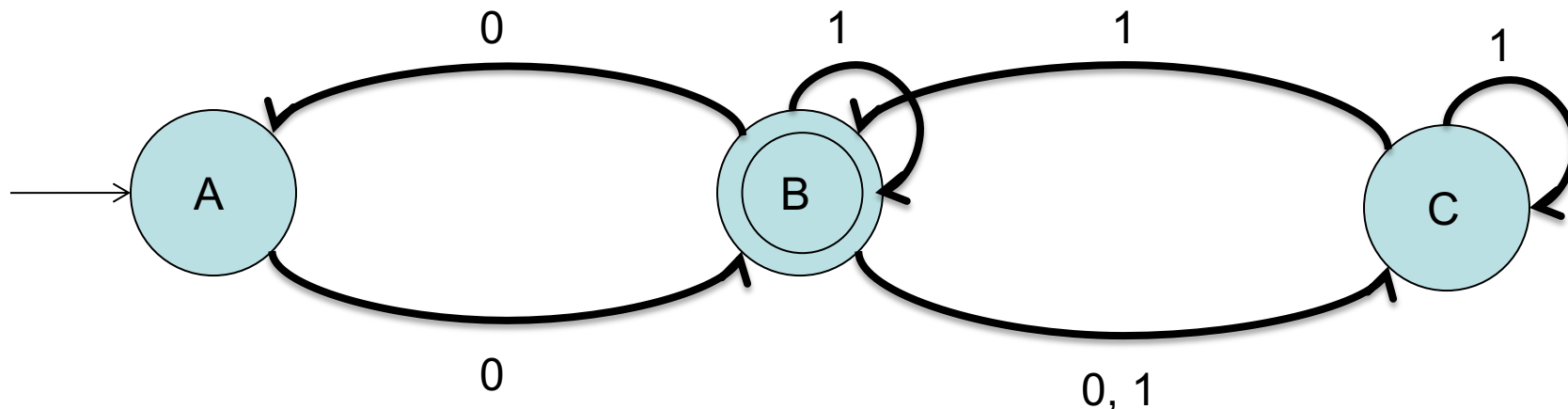
For B, $Q = 0^*1$; $P = B10^*1 + B0 = B(10^*1 + 0)$

- and therefore

- $B = 0^*1(10^*1 + 0)^*$

- Note: This technique fails if there are lambda transitions.

Using Regular Equations



$$A = \lambda + B0$$

$$B = A0 + C1 + B1$$

$$C = B(0+1) + C1; C = B(0+1)1^*$$

$$B = 0 + B00 + B(0+1)1^+ + B1$$

$$B = 0 + B(00 + (0+1)1^+ + 1); B = 0(00 + (0+1)1^+ + 1)^* = 0(1 + (0+1)1^+ + 00)^*$$

This is same form as with state ripping. It won't always be so.

Practice NFAs

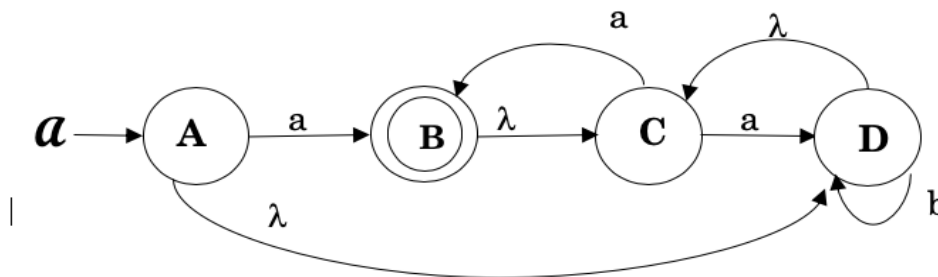
- Write NFAs for each of the following
 - $(111 + 000)^+$
 - $(0+1)^* 101 (0+1)^+$
 - $(1 (0+1)^* 0) + (0 (0+1)^* 1)$
- Convert each NFA you just created to an equivalent DFA.

DFAs to REs

- For each of the DFAs you created for the previous page, use ripping of states and then regular equations to compute the associated regular expression. Note: You obviously ought to get expressions that are equivalent to the initial expressions.

Assignment # 3

1. Write a deterministic finite-state automaton (DFA) that accepts strings over $\{0,1,2\}$. Each string represents a base 3 number read most significant to least significant digit. Accept those representing a number in base 10 that has a remainder of 1 or 3, when divided by 5. Thus, 1, 20, 102, 121 are in (each is $1 \pmod{5}$), and 10, 22, 111, 200 are in (each is $3 \pmod{5}$).
2. Take the two DFAs that I presented on binary strings that are Odd Parity (2 state DFA) and the one that represents decimal numbers with remainders of 2 and 3, when divided by 5, and create the cross-product machine that recognizes all strings that have the intersection of these properties.
3. Use the standard conversion technique (subsets of states) to convert the NFA below to an equivalent DFA. Do not include unreachable states.



Due: Tuesday, Sept. 17, 11:59PM (use Webcourses to turn in)

State Minimization

Minimum State DFAs

State Minimization

- Text makes it an assignment on Page 299 in Edition 2.
- This is too important to defer, IMHO.
- First step is to remove any state that is unreachable from the start state; a depth first search rooted at start state will identify all reachable states
- One seeks to merge compatible states – states q and s are compatible if, for all strings x , $\delta^*(q,x)$ and $\delta^*(s,x)$ are either both an accepting or both rejecting states
- One approach is to discover incompatible states – states q and s are incompatible if there exists a string x such that one of $\delta^*(q,x)$ and $\delta^*(s,x)$ is an accepting state and the other is not
- There are many ways to approach this but my favorite is to do incompatible states via an n by n lower triangular matrix

Sample Minimization

- This uses a transition table
- Just an X denotes Immediately incompatible
- Pairs are dependencies for compatibility
- If a dependent is incompatible, so are pairs that depend on it
- When done, any not x--ed out are compatible
- Here, new states are $\langle 1,3 \rangle$, $\langle 2,4,5 \rangle$, $\langle 6 \rangle$; $\langle 1,3 \rangle$ is start and not accept; others are accept
- Write new diagram

+

	a	b	c
<u>>1</u>	5	2	2
<u>2</u>	1	6	2
<u>3</u>	2	4	5
<u>4</u>	3	6	2
<u>5</u>	3	6	5
<u>6</u>	1	3	4

<u>2</u>	X				
<u>3</u>	2,5 2,4	X			
<u>4</u>	X	1,3	X		
<u>5</u>	X	1,3	X	2,5	
<u>6</u>	X	3,6 X 2,4	X	1,3 3,6 X 2,4	1,3 3,6 X 4,5
	1	<u>2</u>	3	<u>4</u>	<u>5</u>

Closure Properties

Regular Languages

Reversal of Regular Sets

- It is easier to do this with regular sets than with DFAs
- Let E be some arbitrary expression; E^R is formed by
 - Primitives: $\emptyset^R = \emptyset$ $\lambda^R = \lambda$ $a^R = a$
 - Closure:
 - $(A \cdot B)^R = (B^R \cdot A^R)$
 - $(A + B)^R = (A^R + B^R)$
 - $(A^*)^R = (A^R)^*$
- Challenge: How would you do this with FSA models?
 - Start with DFA; change all final to start states; change start to a final state; and reverse edges
 - Note that this creates multiple start states; can create a new start state with λ -transitions to multiple starts

Substitution

- A substitution is a function, f , from each member, a , of an alphabet, Σ , to a language L_a
- Regular languages are closed under substitution of regular languages (i.e., each L_a is regular)
- Easy to prove by replacing each member of Σ in a regular expression for a language L with regular expression for L_a
- A homomorphism is a substitution where each L_a is a single string

Quotient with Regular Sets

- Quotient of two languages B and C, denoted B/C, is defined as $B/C = \{x \mid \exists y \in C \text{ where } xy \in B\}$
- Let B be recognized by DFA $A_B = (Q_B, \Sigma, \delta_B, q_{1B}, F_B)$ and C by $A_C = (Q_C, \Sigma, \delta_C, q_{1C}, F_C)$
- Define the recognizer for B/C by $A_{B/C} = (Q_B \cup Q_B \times Q_C, \Sigma, \delta_{B/C}, q_{1B}, F_B \times F_C)$
 - $\delta_{B/C}(q, a) = \{\delta_B(q, a)\} \quad a \in \Sigma, q \in Q_B$
 - $\delta_{B/C}(q, \lambda) = \{<q, q_{1C}>\} \quad q \in Q_B$
 - $\delta_{B/C}(<q, p>, \lambda) = \{<\delta_B(q, a), \delta_C(p, a)>\} \quad a \in \Sigma, q \in Q_B, p \in Q_C$
- The basic idea is that we simulate B and then randomly decide it has seen x and continue by looking for y, simulating B continuing after x but with C starting from scratch

Quotient Again

- Assume some class of languages, \mathcal{C} , is closed under concatenation, intersection with regular and substitution of members of \mathcal{C} , show \mathcal{C} is closed under Quotient with Regular
- $L/R = \{ x \mid \exists y \in R \text{ where } xy \in L \}$
 - Define $\Sigma' = \{ a' \mid a \in \Sigma \}$
 - Let $h(a) = a; h(a') = \lambda$ where $a \in \Sigma$
 - Let $g(a) = a'$ where $a \in \Sigma$
 - Let $f(a) = \{a, a'\}$ where $a \in \Sigma$
 - $L/R = h(f(L) \cap (\Sigma^* \cdot g(R)))$

Applying Meta Approach

- $\text{INIT}(L) = \{ x \mid \exists y \in \Sigma^* \text{ where } xy \in L \}$
 - $\text{INIT}(L) = h(f(L) \cap (\Sigma^* \cdot g(\Sigma^*)))$
 - Also $\text{INIT}(L) = L / \Sigma^*$
- $\text{LAST}(L) = \{ y \mid \exists x \in \Sigma^* \text{ where } xy \in L \}$
 - $\text{LAST}(L) = h(f(L) \cap (g(\Sigma^*) \cdot \Sigma^*))$
- $\text{MID}(L) = \{ y \mid \exists x, z \in \Sigma^* \text{ where } xyz \in L \}$
 - $\text{MID}(L) = h(f(L) \cap (g(\Sigma^*) \cdot \Sigma^* \cdot g(\Sigma^*)))$
- $\text{EXTERIOR}(L) = \{ xz \mid \exists y \in \Sigma^* \text{ where } xyz \in L \}$
 - $\text{EXTERIOR}(L) = h(f(L) \cap (\Sigma^* \cdot g(\Sigma^*) \cdot \Sigma^*))$

Making Life Easy

- The key in proving closure is to always try to identify the “best” equivalent formal model for regular sets when trying to prove a particular property
- For example, how could you even conceive of proving closure under intersection and complement in regular expression notations?
- Note how much easier quotient is when have closure under concatenation, and substitution and intersection with regular languages than showing in FSA notation

Reachable and Reaching

- $\text{Reachablefrom}(q) = \{ p \mid \exists w \ni \delta(q,w)=p \}$
 - Just do depth first search from q , marking all reachable states. Works for NFA as well.
- $\text{Reachingto}(q) = \{ p \mid \exists w \ni \delta(p,w)=q \}$
 - Do depth first from q , going backwards on transitions, marking all reaching states. Works for NFA as well.

Min and Max

- $\text{Min}(L) = \{ w \mid w \in L \text{ and no proper prefix of } w \text{ is in } L \} = \{ w \mid w \in L \text{ and if } w=xy, x \in \Sigma^*, y \in \Sigma^+ \text{ then } x \notin L \}$
- $\text{Max}(L) = \{ w \mid w \in L \text{ and } w \text{ is not the proper prefix of any word in } L \} = \{ w \mid w \in L \text{ and if } y \in \Sigma^+ \text{ then } wy \notin L \}$
- Examples:
 - $\text{Min}(0(0+1)^*) = \{0\}$
 - $\text{Max}(0(0+1)^*) = \{\}$
 - $\text{Min}(01 + 0 + 10) = \{0, 10\}$
 - $\text{Max}(01 + 0 + 10) = \{01, 10\}$
 - $\text{Min}(\{a^i b^j c^k \mid i \leq k \text{ or } j \leq k\}) = \{a^i b^j c^k \mid i, j \geq 0, k = \min(i, j)\}$
 - $\text{Max}(\{a^i b^j c^k \mid i \leq k \text{ or } j \leq k\}) = \{\}$ because k has no bound
 - $\text{Min}(\{a^i b^j c^k \mid i \geq k \text{ or } j \geq k\}) = \{\lambda\}$
 - $\text{Max}(\{a^i b^j c^k \mid i \geq k \text{ or } j \geq k\}) = \{a^i b^j c^k \mid i, j \geq 0, k = \max(i, j)\}$

Regular Closed under Min

- Assume L is regular then $\text{Min}(L)$ is regular
- Let $L = L(A)$, where $A = (Q, \Sigma, \delta, q_0, F)$ is a DFA with no state unreachable from q_0
- Define $A_{\text{min}} = (Q \cup \{\text{dead}\}, \Sigma, \delta_{\text{min}}, q_0, F)$, where for $a \in \Sigma$
 $\delta_{\text{min}}(q, a) = \delta(q, a)$, if $q \in Q - F$; $\delta_{\text{min}}(q, a) = \text{dead}$, if $q \in F$;
 $\delta_{\text{min}}(\text{dead}, a) = \text{dead}$

The reasoning is that the machine A_{min} accepts only elements in L that are not extensions of shorter strings in L . By making it so transitions from all final states in A_{min} go to the new “dead” state, we guarantee that extensions of accepted strings will not be accepted by this new automaton.

Therefore, Regular Languages are closed under Min.

Regular Closed under Max

- Assume L is regular then $\text{Max}(L)$ is regular
- Let $L = L(A)$, where $A = (Q, \Sigma, \delta, q_0, F)$ is a DFA with no state unreachable from q_0
- Define $A_{\text{max}} = (Q, \Sigma, \delta, q_0, F_{\text{max}})$, where
 $F_{\text{max}} = \{ f \mid f \in F \text{ and } \text{Reachablefrom}^+(f) \cap F = \emptyset \}$
where $\text{Reachablefrom}^+(q) = \{ p \mid \exists w \ni |w| > 0 \text{ and } \delta(q, w) = p \}$

The reasoning is that the machine A_{max} accepts only elements in L that cannot be extended. If there is a non-empty string that leads from some final state f to any final state, including f , then f cannot be final in A_{max} . All other final states can be retained.

The inductive definition of Reachablefrom^+ is:

1. $\text{Reachablefrom}^+(q)$ contains $\{ s \mid \text{there exists an element of } \Sigma, a, \text{ such that } \delta(q, a) = s \}$
2. If s is in $\text{Reachablefrom}^+(q)$ then $\text{Reachablefrom}^+(q)$ contains $\{ t \mid \text{there exists an element of } \Sigma, a, \text{ such that } \delta(s, a) = t \}$
3. No other states are in $\text{Reachablefrom}^+(q)$

Therefore, Regular Languages are closed under Max.

Pumping Lemma for Regular Languages

What is not a Regular Language

Pumping Lemma Concept

- Let $A = (Q, \Sigma, \delta, q_1, F)$ be a DFA, where $Q = \{q_1, q_2, \dots, q_N\}$
- The “pigeon-hole principle” tells us that whenever we visit $N+1$ or more states, we must visit at least one state more than once (loop)
- Any string, w , of length N or greater leads to us making N transitions after visiting the start state, and so we visit at least one state more than once when reading w

Pumping Lemma For Regular

- Theorem: Let L be regular then there exists an $N > 0$ such that, if $w \in L$ and $|w| \geq N$, then w can be written in the form xyz , where $|xy| \leq N$, $|y| > 0$, and for all $i \geq 0$, $xy^i z \in L$
- This means that interesting regular languages (infinite ones) have a very simple self-embedding property that occurs early in long strings

Pumping Lemma Proof

- If L is regular then it is recognized by some DFA, $A=(Q,\Sigma,\delta,q_0,F)$. Let $|Q| = N$ states. For any string w , such that $|w| \geq N$, A must make $N+1$ state visits to consume its first N characters, followed by $|w|-N$ more state visits.
- In its first $N+1$ state visits, A must enter at least one state two or more times.
- Let $w = v_1 \dots v_j \dots v_k \dots v_m$, where $m = |w|$, and $\delta(q_0, v_1 \dots v_j) = \delta(q_0, v_1 \dots v_k)$, $k > j$, and let this state represent the first one repeated while A consumes w .
- Define $x = v_1 \dots v_j$, $y = v_{j+1} \dots v_k$, and $z = v_{k+1} \dots v_m$. Clearly $w = xyz$. Moreover, since $k > j$, $|y| > 0$, and since $k \leq N$, $|xy| \leq N$.
- Since A is deterministic, $\delta(q_0, xy) = \delta(q_0, xy^i)$, for all $i \geq 0$.
- Thus, if $w \in L$, $\delta(q_0, xyz) \in F$, and so $\delta(q_0, xy^i z) \in F$, for all $i \geq 0$.
- Consequently, if $w \in L$, $|w| \geq N$, then w can be written in the form $xy^i z$, where $|xy| \leq N$, $|y| > 0$, and for all $i \geq 0$, $xy^i z \in L$.

Lemma's Adversarial Process

- Assume $L = \{a^n b^n \mid n > 0\}$ is regular
- P.L.: Provides $N > 0$
 - We CANNOT choose N ; that's the P.L.'s job
- Our turn: Choose $a^N b^N \in L$
 - We get to select a string in L
- P.L.: $a^N b^N = xyz$, where $|xy| \leq N$, $|y| > 0$, and for all $i \geq 0$, $xy^i z \in L$
 - We CANNOT choose split, but P.L. is constrained by N
- Our turn: Choose $i = 0$.
 - We have the power here
- P.L.: $a^{N-|y|} b^N \in L$; just a consequence of P.L.
- Our turn: $a^{N-|y|} b^N \notin L$; just a consequence of L 's structure
- CONTRADICTION, so L is NOT regular

xwx is not Regular (PL)

- $L = \{ x w x \mid x, w \in \{a, b\}^+ \}$:
- Assume that L is Regular.
- PL: Let $N > 0$ be given by the Pumping Lemma.
- YOU: Let s be a string, $s \in L$, such that $s = a^N b a a^N b$
- PL: Since $s \in L$ and $|s| \geq N$, s can be split into 3 pieces, $s = xyz$, such that $|xy| \leq N$ and $|y| > 0$ and $\forall i \geq 0 \ x y^i z \in L$
- YOU: Choose $i = 2$
- PL: $x y^2 z = x y y z \in L$
- Thus, $a^{N + |y|} b a a^N b$ would be in L, but this is not so since $N + |y| \neq N$
- We have arrived at a contradiction.
- Therefore L is not Regular.

$a^{\text{Fib}(k)}$ is not Regular (PL)

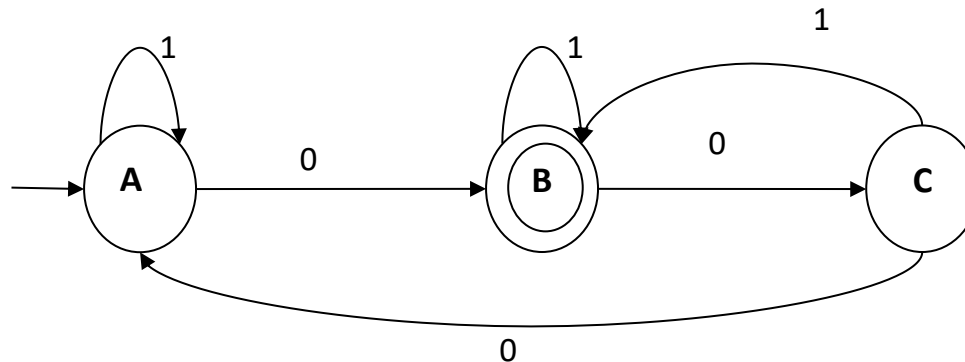
- $L = \{a^{\text{Fib}(k)} \mid k > 0\}$:
- Assume that L is regular
- Let N be the positive integer given by the Pumping Lemma
- Let s be a string $s = a^{\text{Fib}(N+3)} \in L$
- Since $s \in L$ and $|s| \geq N$ ($\text{Fib}(N+3) > N$ in all cases; actually $\text{Fib}(N+2) > N$ as well), s is split by PL into xyz, where $|xy| \leq N$ and $|y| > 0$ and for all $i \geq 0$, $xy^iz \in L$
- We choose $i = 2$; by PL: $xy^2z = xyyz \in L$
- Thus, $a^{\text{Fib}(N+3)+|y|}$ would be $\in L$. This means that there is a Fibonacci number between $\text{Fib}(N+3)$ and $\text{Fib}(N+3)+N$, but the smallest Fibonacci greater than $\text{Fib}(N+3)$ is $\text{Fib}(N+3)+\text{Fib}(N+2)$ and $\text{Fib}(N+2) > N$
This is a contradiction; therefore L is not regular ■
- Note: Using values less than $N+3$ could be dangerous because N could be 1 and both $\text{Fib}(2)$ and $\text{Fib}(3)$ are within N (1) of $\text{Fib}(1)$.

Pumping Lemma Problems

- Use the Pumping Lemma to show each of the following is not regular
 - $\{ 0^m 1^{2n} \mid m \leq n \}$
 - $\{ ww^R \mid w \in \{a,b\}^+ \}$
 - $\{ 1^{n^2} \mid n > 0 \}$
 - $\{ ww \mid w \in \{a,b\}^+ \}$

Assignment # 4.1

1. Convert the DFA below to a regular expression, first by using either the GNFA (or state ripping) or the R_{ij}^k approach, and then by using regular equations. You must show all steps in each part of this solution.



Due: Tuesday, Sept. 24, 11:59PM (Use Webcourses to turn in)

Assignment # 4.2

2. Minimize the number of states in the following DFA, showing the determination of incompatible states (table on right).

	a	b	c
>1	2	3	6
2	5	4	4
<u>3</u>	1	4	5
<u>4</u>	6	3	5
5	5	2	4
6	2	4	1

2					
<u>3</u>					
<u>4</u>					
5					
6					
	>1	2	<u>3</u>	<u>4</u>	5

Construct and write down your new, equivalent automaton!!
Due: Tuesday, Sept. 24, 11:59PM (use Webcourses to turn in)

State Minimization

Minimum State DFAs

Myhill-Nerode Theorem

Myhill-Nerode Theorem

The following are equivalent:

1. L is accepted by some DFA
2. L is the union of some of the classes of a right invariant equivalence relation, R , of finite index.
3. The specific right invariance equivalence relation R_L where $x R_L y$ iff $\forall z [xz \in L \text{ iff } yz \in L]$ has finite index

Definition. R is a right invariant equivalence relation iff R is an equivalence relation and $\forall z [x R y \text{ implies } xz R yz]$.

Note: This is only meaningful for relations over strings.

Myhill-Nerode 1 \Rightarrow 2

1. Assume L is accepted by some DFA, $A = (Q, \Sigma, \delta, q_1, F)$
2. Define R_A by $x R_A y$ iff $\delta^*(q_1, x) = \delta^*(q_1, y)$. First, R_A is defined by equality and so is obviously an equivalence relation (Clearly if $\delta^*(q_1, x) = \delta^*(q_1, y)$ then $\forall z \delta^*(q_1, xz) = \delta^*(q_1, yz)$ because A is deterministic. Moreover if $\forall z \delta^*(q_1, xz) = \delta^*(q_1, yz)$ then $\delta^*(q_1, x) = \delta^*(q_1, y)$, just by letting $z = \lambda$. Putting it together $x R_A y \iff \forall z xz R_A yz$. Thus, R_A is right invariant; its index is $|Q|$ which is finite; and $L(A) = \bigcup_{\delta^*(x) \in F} [x]_{R_A}$, where $[x]_{R_A}$ refers to the equivalence class containing the string x .

Myhill-Nerode 2 \Rightarrow 3

2. Assume L is the union of some of the classes of a right invariant equivalence relation, R , of finite index.
3. Since $x R y$ iff $\forall z [xz R yz]$, R is right invariant and L is the union of some of the equivalence classes, then $x R y \Rightarrow \forall z [xz \in L \text{ iff } yz \in L] \Rightarrow x R_L y$.
This means that the index of R_L is less than or equal to that of R and so is finite. Note that the index of R_L is then less than or equal to that of any other right invariant equivalence relation, R , of finite index that defines L .

Myhill-Nerode 3 \Rightarrow 1

3. Assume the specific right invariance equivalence relation R_L where $x R_L y$ iff $\forall z [xz \in L \text{ iff } yz \in L]$ has finite index
1. Define the automaton $A = (Q, \Sigma, \delta, q_1, F)$ by
 - $Q = \{ [x]_{R_L} \mid x \in \Sigma^* \}$
 - $\delta([x]_{R_L}, a) = [xa]_{R_L}$
 - $q_1 = [\lambda]$
 - $F = \{ [x]_{R_L} \mid x \in L \}$

Note: This is the minimum state automaton and all others are either equivalent or have redundant indistinguishable states

Use of Myhill-Nerode

- $L = \{a^n b^n \mid n > 0\}$ is NOT regular.
- Assume otherwise.
- M-N says that the specific r.i. equiv. relation R_L has finite index, where $x R_L y$ iff $\forall z [xz \in L \text{ iff } yz \in L]$.
- Consider the equivalence classes $[a^i b]$ and $[a^j b]$, where $i, j > 0$ and $i \neq j$.
- $a^i b b^{i-1} \in L$ but $a^j b b^{i-1} \notin L$ and so $[a^i b]$ is not related to $[a^j b]$ under R_L and thus $[a^i b] \neq [a^j b]$.
- This means that R_L has infinite index.
- Therefore L is not regular.

xwx is not Regular (MN)

- **$L = \{ x a x \mid x \in \{a,b\}^+ \}$:**
- We consider the right invariant equivalence class $[a^i b]$, $i > 0$.
- It's clear that $a^i b a a^i b$ is in the language, but $a^k b a a^i b$ is not when $k < i$.
- This shows that there is a separate equivalence class, $[a^i b]$, induced by R_L , for each $i > 0$. Thus, the index of R_L is infinite and Myhill-Nerode states that L cannot be Regular.

$a^{\text{Fib}(k)}$ is not Regular (MN)

- $L = \{a^{\text{Fib}(k)} \mid k > 0\}$:
- We consider the collection of right invariant equivalence classes $[a^{\text{Fib}(j)}]$, $j > 2$.
- It's clear that $a^{\text{Fib}(j)}a^{\text{Fib}(j+1)}$ is in the language, but $a^{\text{Fib}(k)}a^{\text{Fib}(j+1)}$ is not when $k > 2$ and $k \neq j$ and $k \neq j+2$
- This shows that there is a separate equivalence class $[a^{\text{Fib}(j)}]$ induced by R_L , for each $j > 2$.
- Thus, the index of R_L is infinite and Myhill-Nerode states that L cannot be Regular.

Myhill-Nerode and Minimization

- Corollary: The minimum state DFA for a regular language, L , is formed from the specific right invariance equivalence relation R_L where
 $x R_L y$ iff $\forall z [xz \in L \text{ iff } yz \in L]$
- Moreover, all minimum state machines have the same structure as the above, except perhaps for the names of states

What is Regular So Far?

- Any language accepted by a DFA
- Any language accepted by an NFA
- Any language specified by a Regular Expression
- Any language representing the unique solution to a set of properly constrained regular equations

What is NOT Regular?

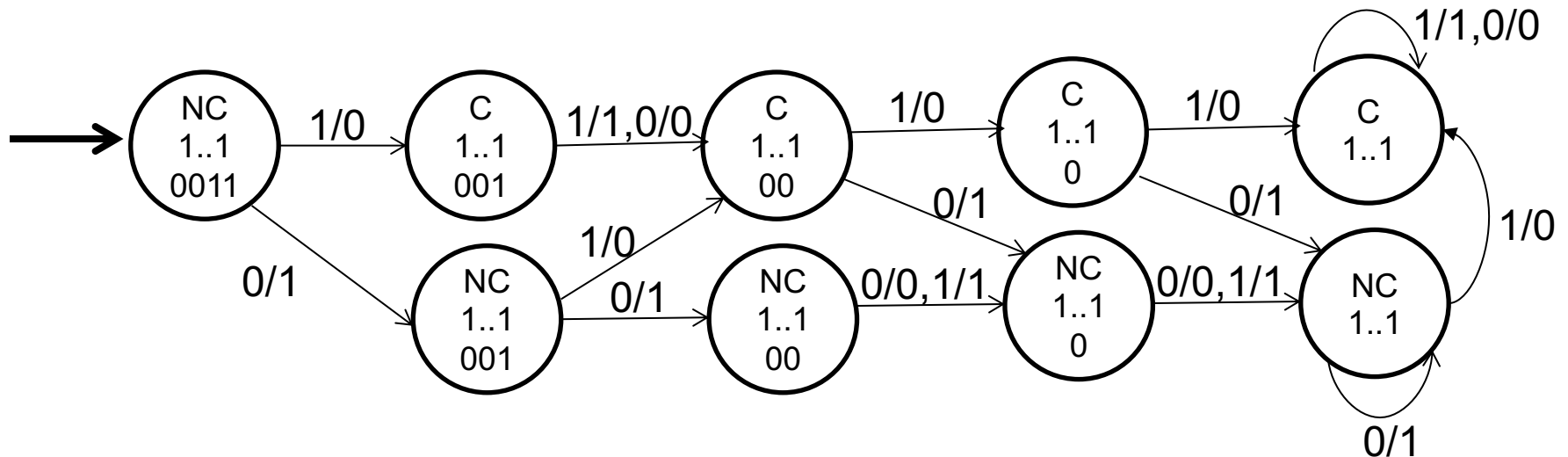
- Well, anything for which you cannot write an accepting DFA or NFA, or a defining regular expression, or a right/left linear grammar, or a set of regular equations, but that's not a very useful statement
- There are two tools we have:
 - Pumping Lemma for Regular Languages
 - Myhill-Nerode Theorem

Finite-State Transducers

- A transducer is a machine with output
- Mealy Model
 - $M = (Q, \Sigma, \Gamma, \delta, \gamma, q_0)$
 - Γ is the finite output alphabet
 - $\gamma: Q \times \Sigma \rightarrow \Gamma$ is the output function
 - Essentially a Mealy Model machine produced a character of output for each character of input it consumes, and it does so on the transitions from one state to the next.
 - A Mealy Model represents a synchronous circuit whose output is triggered each time a new input arrives.

Sample Mealy Model

- Write a Mealy finite-state machine that produces the 2's complement result of subtracting 1101 from a binary input stream (assuming at least 4 bits of input)



Finite-State Transducers

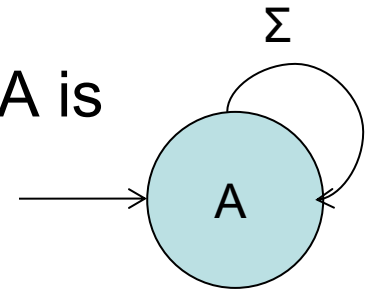
- Moore Model
 - $M = (Q, \Sigma, \Gamma, \delta, \gamma, q_0)$
 - Γ is the finite output alphabet
 - $\gamma: Q \rightarrow \Gamma$ is the output function
 - Essentially a Moore Model machine produced a character of output whenever it enters a state, independent of how it arrived at that state.
 - A Moore Model represents an asynchronous circuit whose output is a steady state until new input arrives.

Summary of Decision and Closure Properties

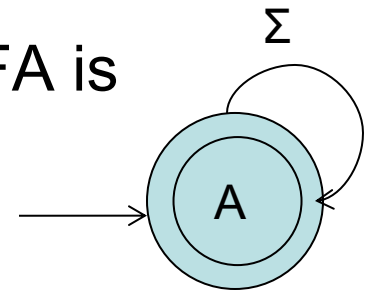
Regular Languages

Decidable Properties

- Membership (just run DFA over string)
- $L = \emptyset$: Minimize and see if minimum state DFA is



- $L = \Sigma^*$: Minimize and see if minimum state DFA is



- Finiteness: Minimize and see if there are no loops emanating on a path to a final state
- Equivalence: Minimize both and see if isomorphic

Closure Properties

- Virtually everything with members of its own class as we have already shown
- Union, concatenation, Kleene *, complement, intersection, set difference, reversal, substitution, homomorphism, quotient with regular sets, Prefix, Suffix, Substring, Exterior, Min, Max and so much more

Regular Languages # 1

- Finite Automata
- Moore and Mealy models: Automata with output.
- Regular operations
- Non-determinism: Its use. Conversion to deterministic FSAs. Formal proof of equivalence.
- Lambda moves: Lambda closure of a state
- Regular expressions
- Equivalence of REs and FSAs.
- Pumping Lemma: Proof and applications.

Regular Languages # 2

- Regular equations: REQs and FSAs.
- Myhill-Nerode Theorem: Right invariant equivalence relations. Specific relation for a language L . Proof and applications.
- Minimization: Why it's unique. Process of minimization. Analysis of cost of different approaches.
- Regular (right linear) grammars, regular languages and their equivalence to FSA languages.

Regular Languages # 3

- Closure properties: Union, concatenation, Kleene *, complement, intersection, set difference, reversal, substitution, homomorphism and quotient with regular sets, Prefix, Suffix, Substring, Exterior.
- Algorithms for reachable states and states that can reach some other chosen states.
- Decision properties: Emptiness, finiteness, equivalence.

Assignment # 5

1. For each of the following, prove it is not regular by using the Pumping Lemma or Myhill-Nerode. You must do at least one of these using the Pumping Lemma and at least one using Myhill-Nerode.
 - a. $L = \{ a^i b^j \mid i > 2*j \}$
 - b. $L = \{ a^{f(n)} \mid f(0) = 2; f(i+1) = f(i)^2 \}$
 - c. $L = \{ a^{g(i)} \mid g(i) = i^3 \}$
2. Write a regular (right linear) grammar that generates $L =$ the regular set represented by $(00 + 010 + 001)^*$.
3. Present a Mealy Model finite state machine that reads an input $x \in \{0, 1\}^*$ and produces the binary number that represents the result of adding binary **10101** to x (assumes all numbers are positive, including results). Note: The binary number is read from least to most significant bit.

Due: Tuesday, Oct. 1, 11:59PM (use Webcourses to turn in)

Formal Languages

Includes and Expands on
Chapter 2 of Sipser

History of Formal Language

- In 1940s, Emil Post (mathematician) devised rewriting systems as a way to describe how mathematicians do proofs. Purpose was to mechanize them.
- Early 1950s, Noam Chomsky (linguist) developed a hierarchy of rewriting systems (grammars) to describe natural languages.
- Late 1950s, Backus-Naur (computer scientists) devised BNF (a variant of Chomsky's context-free grammars) to describe the programming language Algol.
- 1960s was the time of many advances in parsing. In particular, parsing of context free was shown to be no worse than $O(n^3)$. More importantly, useful subsets were found that could be parsed in $O(n)$.

Grammars

- $G = (V, \Sigma, R, S)$ is a Phrase Structured Grammar (PSG) where
 - V : Finite set of non-terminal symbols
 - Σ : Finite set of terminal symbols
 - R : finite set of rules of form $\alpha \rightarrow \beta$,
 - α in $(V \cup \Sigma)^* V (V \cup \Sigma)^*$
 - β in $(V \cup \Sigma)^*$
 - S : a member of V called the start symbol
- Right linear restricts all rules to be of forms
 - α in V
 - β of form $\Sigma V, \Sigma$ or λ

Derivations

- $x \Rightarrow y$ reads as x derives y iff
 - $x = \gamma\alpha\delta$, $y = \gamma\beta\delta$ and $\alpha \rightarrow \beta$
- \Rightarrow^* is the reflexive, transitive closure of \Rightarrow
- \Rightarrow^+ is the transitive closure of \Rightarrow
- $x \Rightarrow^* y$ iff $x = y$ or $x \Rightarrow^* z$ and $z \Rightarrow y$
- Or, $x \Rightarrow^* y$ iff $x = y$ or $x \Rightarrow z$ and $z \Rightarrow^* y$
- $L(G) = \{ w \mid S \Rightarrow^* w \}$ is the language generated by G .

Regular Grammars

- Regular grammars are also called right linear grammars
- Each rule of a regular grammar is constrained to be of one of the three forms:

$$A \rightarrow \lambda, \quad A \in V$$

$$A \rightarrow a, \quad A \in V, a \in \Sigma$$

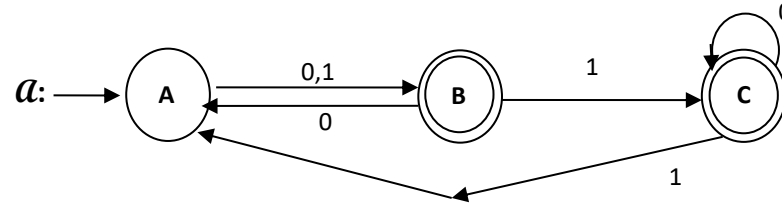
$$A \rightarrow aB, \quad A, B \in V, a \in \Sigma$$

DFA to Regular Grammar

- Every language recognized by a DFA is generated by an equivalent regular grammar
- Given $A = (Q, \Sigma, \delta, q_0, F)$, $L(A)$ is generated by $G_A = (Q, \Sigma, R, q_0)$ where R contains
$$q \rightarrow as \quad \text{iff } \delta(q, a) = s, a \in \Sigma$$
$$q \rightarrow \lambda \quad \text{iff } q \in F$$

Example of DFA to Grammar

- **DFA**



- **Grammar**

A → **0 B** | **1 B**

B → **0 A** | **1 C** | λ

C → **0 C** | **1 A** | λ

Regular Grammar to NFA

- Every language generated by a regular grammar is recognized by an equivalent NFA
- Given $G = (V, \Sigma, R, S)$, $L(G)$ is recognized by $A_G = (V \cup \{f\}, \Sigma, \delta, S, \{f\})$ where δ is defined by
 - $\delta(A, a) \subseteq \{B\}$ iff $A \rightarrow aB$
 - $\delta(A, a) \subseteq \{f\}$ iff $A \rightarrow a$
 - $\delta(A, \lambda) \subseteq \{f\}$ iff $A \rightarrow \lambda$

Example of Grammar to NFA

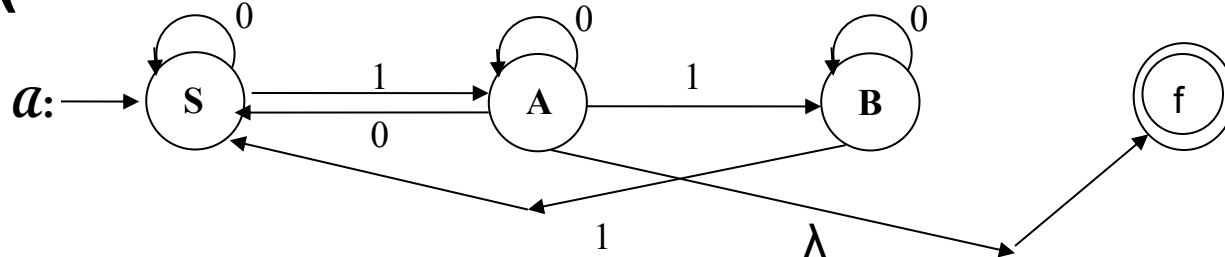
- Grammar

S → **0 S** | **1 A**

A → **0 S** | **0 A** | **1 B** | λ

B → **1 S** | **0 B**

- NFA (can remove f and make A final)



What More is Regular?

- Any language, L , generated by a right linear grammar
- Any language, L , generated by a left linear grammar ($A \rightarrow a, A \rightarrow \lambda, A \rightarrow Ba$)
 - Easy to see L is regular as we can reverse these rules and get a right linear grammar that generates L^R , but then L is the reverse of a regular language which is regular
 - Similarly, the reverse L^R of any regular language L is right linear and hence the language itself is left linear
- Any language, L , that is the union of some of the classes of a right invariant equivalence relation of finite index

Mixing Right and Left Linear

- We can get non-Regular languages if we present grammars that have both right and left linear rules
- To see this, consider $G = (\{S,T\}, \Sigma, R, S)$, where R is:
 - $S \rightarrow aT$
 - $T \rightarrow Sb \mid b$
- $L(G) = \{ a^n b^n \mid n > 0 \}$ which is a classic non-regular, context-free language

Context Free Languages

Context Free Grammar

$G = (V, \Sigma, R, S)$ is a PSG where

Each member of R is of the form

$A \rightarrow \alpha$ where α is a strings $(V \cup \Sigma)^*$

Note that the left hand side of a rule is a letter in V ;

The right hand side is a string from the combined alphabets

The right hand side can even be empty (ε or λ)

A context free grammar is denoted as a CFG and the language generated is a Context Free Language (CFL).

A CFL is recognized by a Push Down Automaton (PDA) to be discussed a bit later.

Sample CFG

Example of a grammar for a small language:

$G = (\{\langle \text{program} \rangle, \langle \text{stmt-list} \rangle, \langle \text{stmt} \rangle, \langle \text{expression} \rangle\},$
 $\{\text{begin, end, ident, ;, =, +, -}\}, R, \langle \text{program} \rangle)$ where R is

$\langle \text{program} \rangle \rightarrow \text{begin } \langle \text{stmt-list} \rangle \text{ end}$

$\langle \text{stmt-list} \rangle \rightarrow \langle \text{stmt} \rangle \mid \langle \text{stmt} \rangle ; \langle \text{stmt-list} \rangle$

$\langle \text{stmt} \rangle \rightarrow \text{ident} = \langle \text{expression} \rangle$

$\langle \text{expression} \rangle \rightarrow \text{ident} + \text{ident} \mid \text{ident} - \text{ident} \mid \text{ident}$

Here “ident” is a token return from a scanner, as are “begin”, “end”, “;”, “=”, “+”, “-”

Note that “;” is a separator (Pascal style) not a terminator (C style).

Derivation

A sentence generation is called a derivation.

Grammar for a simple assignment statement:

R1 $\langle \text{assgn} \rangle \rightarrow \langle \text{id} \rangle := \langle \text{expr} \rangle$
R2 $\langle \text{id} \rangle \rightarrow a \mid b \mid c$
R3 $\langle \text{expr} \rangle \rightarrow \langle \text{id} \rangle + \langle \text{expr} \rangle$
R4 $\quad \quad \quad | \langle \text{id} \rangle * \langle \text{expr} \rangle$
R5 $\quad \quad \quad | (\langle \text{expr} \rangle)$
R6 $\quad \quad \quad | \langle \text{id} \rangle$

In a **leftmost derivation** only the leftmost non-terminal is replaced

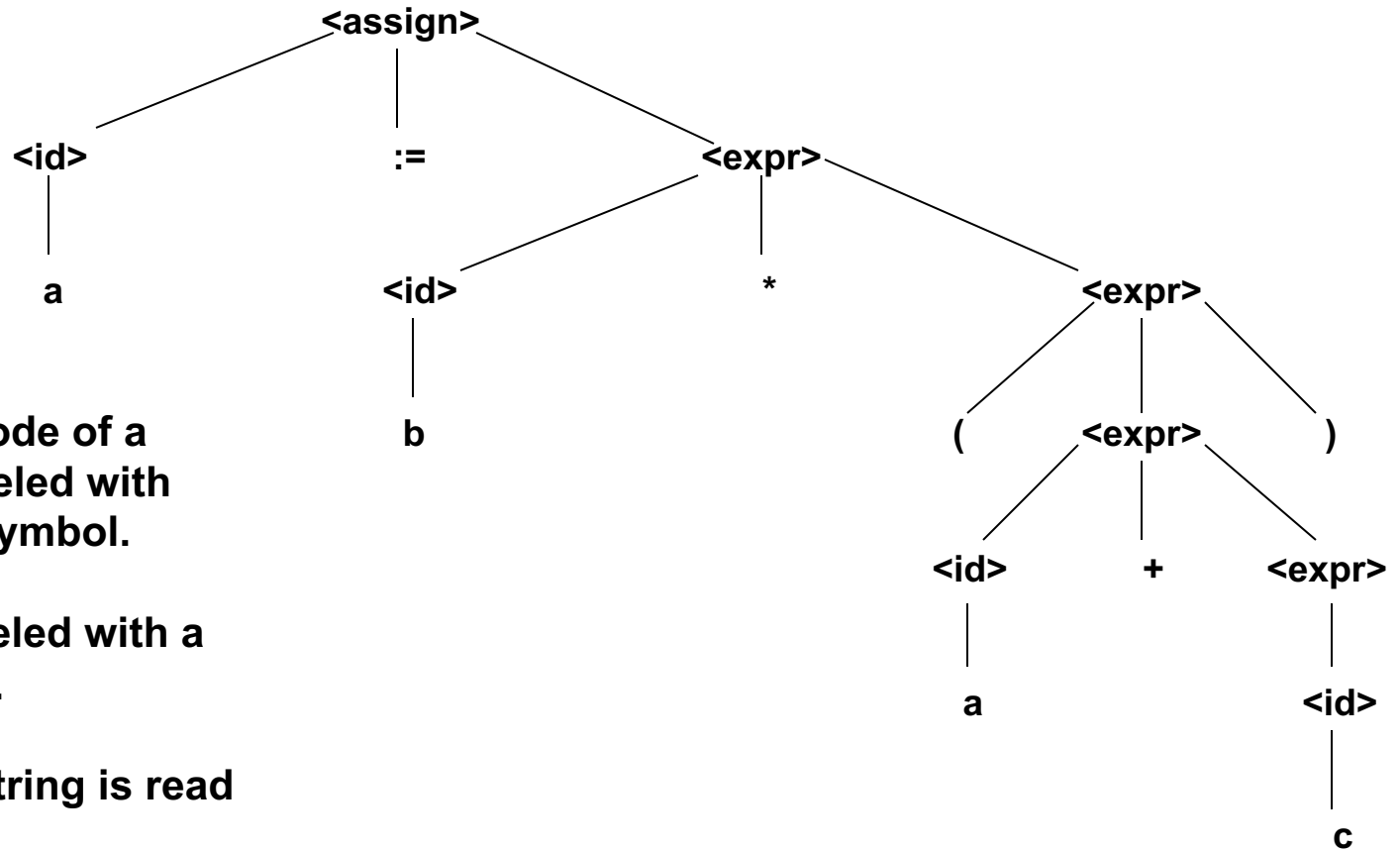
The statement $a := b * (a + c)$
Is generated by the **leftmost derivation**:

$\langle \text{assgn} \rangle \Rightarrow \langle \text{id} \rangle := \langle \text{expr} \rangle$ R1
 $\Rightarrow a := \langle \text{expr} \rangle$ R2
 $\Rightarrow a := \langle \text{id} \rangle * \langle \text{expr} \rangle$ R4
 $\Rightarrow a := b * \langle \text{expr} \rangle$ R2
 $\Rightarrow a := b * (\langle \text{expr} \rangle)$ R5
 $\Rightarrow a := b * (\langle \text{id} \rangle + \langle \text{expr} \rangle)$ R3
 $\Rightarrow a := b * (a + \langle \text{expr} \rangle)$ R2
 $\Rightarrow a := b * (a + \langle \text{id} \rangle)$ R6
 $\Rightarrow a := b * (a + c)$ R2

Parse Trees

A parse tree is a graphical representation of a derivation

For instance the parse tree for the statement $a := b * (a + c)$ is:



Every internal node of a parse tree is labeled with a non-terminal symbol.

Every leaf is labeled with a terminal symbol.

The generated string is read left to right

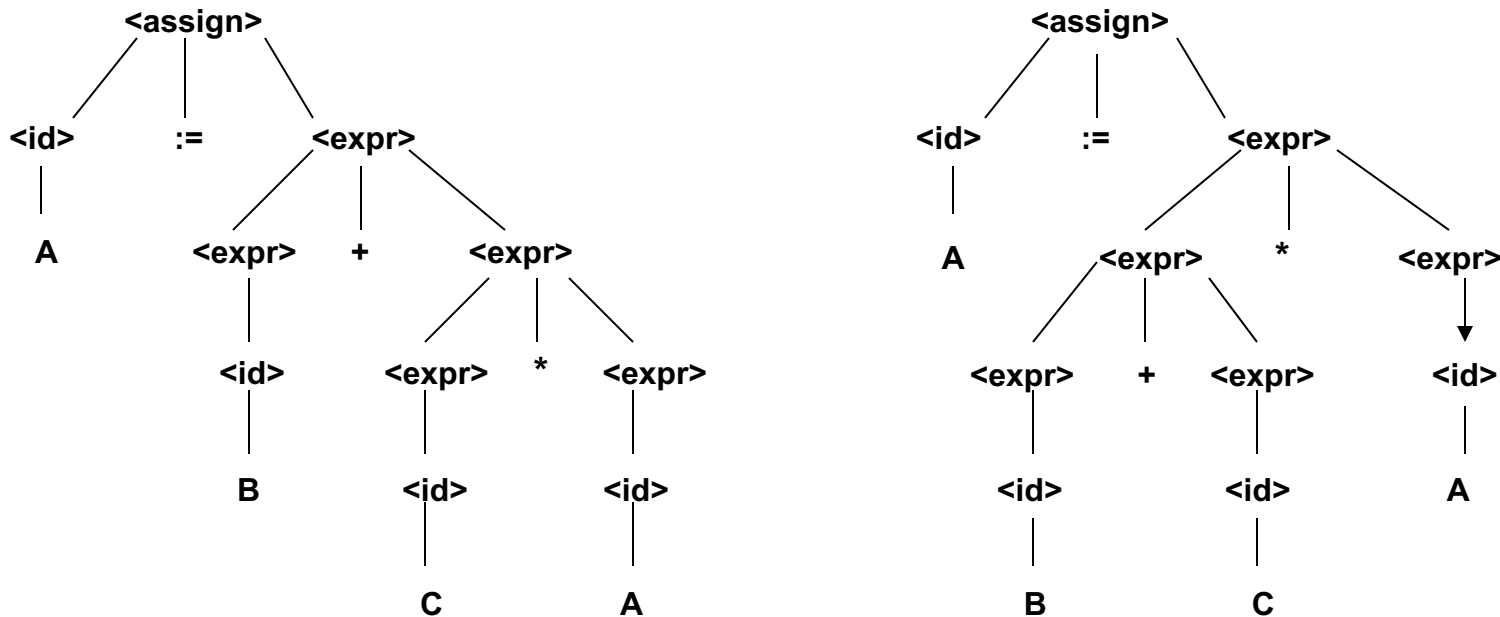
Ambiguity

A grammar that generates a sentence for which there are two or more distinct parse trees is said to be “ambiguous”

For instance, the following grammar is ambiguous because it generates distinct parse trees for the expression $a := b + c * a$

```
<assgn> → <id> := <expr>
<id>    → a | b | c
<expr>  → <expr> + <expr>
          | <expr> * <expr>
          | ( <expr> )
          | <id>
```

Ambiguous Parse



This grammar generates two parse trees for the same expression.

If a language structure has more than one parse tree, the meaning of the structure cannot be determined uniquely.

Precedence

Operator precedence:

If an operator is generated lower in the parse tree, it indicates that the operator has precedence over the operator generated higher up in the tree.

An unambiguous grammar for expressions:

$\langle \text{assign} \rangle \rightarrow \langle \text{id} \rangle := \langle \text{expr} \rangle$
 $\langle \text{id} \rangle \rightarrow a \mid b \mid c$
 $\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle$
 $\mid \langle \text{term} \rangle$
 $\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle * \langle \text{factor} \rangle$
 $\mid \langle \text{factor} \rangle$
 $\langle \text{factor} \rangle \rightarrow (\langle \text{expr} \rangle)$
 $\mid \langle \text{id} \rangle$

This grammar indicates the usual precedence order of multiplication and addition operators.

This grammar generates unique parse trees independently of doing a rightmost or leftmost derivation

Left (right)most Derivations

Leftmost derivation:

$\langle \text{assgn} \rangle \rightarrow \langle \text{id} \rangle := \langle \text{expr} \rangle$
 $\rightarrow a := \langle \text{expr} \rangle$
 $\rightarrow a := \langle \text{expr} \rangle + \langle \text{term} \rangle$
 $\rightarrow a := \langle \text{term} \rangle + \langle \text{term} \rangle$
 $\rightarrow a := \langle \text{factor} \rangle + \langle \text{term} \rangle$
 $\rightarrow a := \langle \text{id} \rangle + \langle \text{term} \rangle$
 $\rightarrow a := b + \langle \text{term} \rangle$
 $\rightarrow a := b + \langle \text{term} \rangle * \langle \text{factor} \rangle$
 $\rightarrow a := b + \langle \text{factor} \rangle * \langle \text{factor} \rangle$
 $\rightarrow a := b + \langle \text{id} \rangle * \langle \text{factor} \rangle$
 $\rightarrow a := b + c * \langle \text{factor} \rangle$
 $\rightarrow a := b + c * \langle \text{id} \rangle$
 $\rightarrow a := b + c * a$

Rightmost derivation:

$\langle \text{assgn} \rangle \Rightarrow \langle \text{id} \rangle := \langle \text{expr} \rangle$
 $\Rightarrow \langle \text{id} \rangle := \langle \text{expr} \rangle + \langle \text{term} \rangle$
 $\Rightarrow \langle \text{id} \rangle := \langle \text{expr} \rangle + \langle \text{term} \rangle * \langle \text{factor} \rangle$
 $\Rightarrow \langle \text{id} \rangle := \langle \text{expr} \rangle + \langle \text{term} \rangle * \langle \text{id} \rangle$
 $\Rightarrow \langle \text{id} \rangle := \langle \text{expr} \rangle + \langle \text{term} \rangle * a$
 $\Rightarrow \langle \text{id} \rangle := \langle \text{expr} \rangle + \langle \text{factor} \rangle * a$
 $\Rightarrow \langle \text{id} \rangle := \langle \text{expr} \rangle + \langle \text{id} \rangle * a$
 $\Rightarrow \langle \text{id} \rangle := \langle \text{expr} \rangle + c * a$
 $\Rightarrow \langle \text{id} \rangle := \langle \text{term} \rangle + c * a$
 $\Rightarrow \langle \text{id} \rangle := \langle \text{factor} \rangle + c * a$
 $\Rightarrow \langle \text{id} \rangle := \langle \text{id} \rangle + c * a$
 $\Rightarrow \langle \text{id} \rangle := b + c * a$
 $\Rightarrow a := b + c * a$

Ambiguity Test

- A Grammar is Ambiguous if there are two distinct parse trees for some string
- Or, two distinct leftmost derivations
- Or, two distinct rightmost derivations
- Some languages are inherently ambiguous but many are not
- Unfortunately (to be shown later) there is no systematic test for ambiguity of context free grammars

Unambiguous Grammar

When we encounter ambiguity, we try to rewrite the grammar to avoid ambiguity.

The ambiguous expression grammar:

$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle \mid \text{id} \mid \text{int} \mid (\langle \text{expr} \rangle)$

$\langle \text{op} \rangle \rightarrow + \mid - \mid * \mid /$

Can be rewritten as:

$\langle \text{expr} \rangle \rightarrow \langle \text{term} \rangle \mid \langle \text{expr} \rangle + \langle \text{term} \rangle \mid \langle \text{expr} \rangle - \langle \text{term} \rangle$

$\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle \mid \langle \text{term} \rangle * \langle \text{factor} \rangle \mid \langle \text{term} \rangle / \langle \text{factor} \rangle.$

$\langle \text{factor} \rangle \rightarrow \text{id} \mid \text{int} \mid (\langle \text{expr} \rangle)$

Parsing Problem

The parsing Problem: Take a string of symbols in a language (tokens) and a grammar for that language to construct the parse tree or report that the sentence is syntactically incorrect.

For correct strings:

Sentence + grammar \rightarrow parse tree

For a compiler, a sentence is a program:

Program + grammar \rightarrow parse tree

Types of parsers:

Top-down aka predictive (recursive descent parsing)

Bottom-up aka shift-reduce

Inherent Ambiguity

- There are some CFLs that are inherently ambiguous and others for which we may just have carelessly written an ambiguous grammar.
- We will see later in course that it is not possible to inspect an arbitrary CFG and determine if it is unambiguous.
- However, parsers must be unambiguous to avoid semantic ambiguity.

Not All is Lost

- Just because we cannot determine ambiguity of a grammar does not mean we cannot have a subclass of grammars that are guaranteed to be unambiguous and that can be used to generate precisely the set of unambiguous CFLs.

LR(k) and LL(k) Grammars

- An LL(k) grammar is a grammar that can drive a top-down parse by always making the right parsing decision with just k tokens of lookahead.
- An LR(k) grammar is a grammar that can drive a bottom-up parse by always making the right parsing decision with just k tokens of lookahead.

LL(k) Grammars

- LL means reading the input from left-to-right using a leftmost derivation with a correct decision requiring just k tokens of lookahead.
- There is an algorithm to determine, for any given k , whether an arbitrary CFG is LL(k).
- LL($k+1$) grammars can generate languages that cannot be generated by LL(k) ones.
- $\lim_{k \rightarrow \infty} \text{LL}(k)$ gets all unambiguous CFLs.
- All programming languages you work with are LL(1) so long as we cheat and use a symbol table.
- LL parsers hate left recursion

LR(k) Grammars

- LR means reading the input from left-to-right using a right derivation run in reverse with a correct decision requiring just k tokens of lookahead.
- There is an algorithm to determine, for any given k , whether an arbitrary CFG is LR(k).
- LR(1) grammars are sufficient to generate to any and all unambiguous CFLs.
- All programming languages you work with are LR(1) so long as we cheat and use a symbol table.
- LR parsers hate right recursion.

Removing Left Recursion if doing Top Down

Given left recursive and non left recursive rules

$$A \rightarrow A\alpha_1 \mid \dots \mid A\alpha_n \mid \beta_1 \mid \dots \mid \beta_m$$

Can view as

$$A \rightarrow (\beta_1 \mid \dots \mid \beta_m) (\alpha_1 \mid \dots \mid \alpha_n)^*$$

Star notation is an extension to normal notation with obvious meaning

Now, it should be clear this can be done right recursive as

$$A \rightarrow \beta_1 B \mid \dots \mid \beta_m B$$

$$B \rightarrow \alpha_1 B \mid \dots \mid \alpha_n B \mid \lambda$$

Right Recursive Expressions

Grammar: $\text{Expr} \rightarrow \text{Expr} + \text{Term} \mid \text{Term}$
 $\text{Term} \rightarrow \text{Term} * \text{Factor} \mid \text{Factor}$
 $\text{Factor} \rightarrow (\text{Expr}) \mid \text{Int}$

Fix: $\text{Expr} \rightarrow \text{Term ExprRest}$
 $\text{ExprRest} \rightarrow + \text{Term ExprRest} \mid \lambda$
 $\text{Term} \rightarrow \text{Factor TermRest}$
 $\text{TermRest} \rightarrow * \text{Factor TermRest} \mid \lambda$
 $\text{Factor} \rightarrow (\text{Expr}) \mid \text{Int}$

Bottom Up vs Top Down

- Bottom-Up: Two stack operations
 - Shift (move input symbol to stack)
 - Reduce (replace top of stack α with A , when $A \rightarrow \alpha$)
 - Challenge is when to do shift or reduce and what reduce to do.
 - Can have both kinds of conflict
- Top-Down:
 - If top of stack is terminal
 - If same as input, read and pop
 - If not, we have an error
 - If top of stack is a non-terminal A
 - Replace A with some α , when $A \rightarrow \alpha$
 - Challenge is what A -rule to use

Chomsky Normal Form

- Each rule of a CFG is constrained to be of one of the three forms:

$$A \rightarrow a, \quad A \in V, a \in \Sigma$$

$$A \rightarrow BC, \quad A, B, C \in V$$

- If the language contains λ then we allow

$$S \rightarrow \lambda$$

and constrain all non-terminating rules of form to be

$$A \rightarrow BC, \quad A \in V, B, C \in V - \{S\}$$

Nullable Symbols

- Let $G = (V, \Sigma, R, S)$ be an arbitrary CFG
- Compute the set $\text{Nullable}(G) = \{A \mid A \Rightarrow^* \lambda\}$
- $\text{Nullable}(G)$ is computed as follows
 $\text{Nullable}(G) \supseteq \{A \mid A \rightarrow \lambda\}$
Repeat
 $\text{Nullable}(G) \supseteq \{B \mid B \rightarrow \alpha \text{ and } \alpha \in \text{Nullable}^*\}$
until no new symbols are added

Removal of λ -Rules

- Let $G = (V, \Sigma, R, S)$ be an arbitrary CFG
- Compute the set $\text{Nullable}(G)$
- Remove all λ -rules
- For each rule of form $B \rightarrow \alpha A \beta$ where A is nullable, add in the rule $B \rightarrow \alpha \beta$
- The above has the potential to greatly increase the number of rules and add unit rules (those of form $B \rightarrow C$, where $B, C \in V$)
- If S is nullable, add new start symbol S_0 , as new start state, plus rules $S_0 \rightarrow \lambda$ and $S_0 \rightarrow \alpha$, where $S \rightarrow \alpha$

Chains (Unit Rules)

- Let $G = (V, \Sigma, R, S)$ be an arbitrary CFG that has had its λ -rules removed
- For $A \in V$, $\text{Chain}(A) = \{ B \mid A \Rightarrow^* B, B \in V \}$
- $\text{Chain}(A)$ is computed as follows
 $\text{Chain}(A) \ni \{ A \}$
Repeat
 $\text{Chain}(A) \ni \{ C \mid B \rightarrow C \text{ and } B \in \text{Chain}(A) \}$
until no new symbols are added

Removal of Unit-Rules

- Let $G = (V, \Sigma, R, S)$ be an arbitrary CFG that has had its λ -rules removed, except perhaps from start symbol
- Compute $\text{Chain}(A)$ for all $A \in V$
- Create the new grammar $G = (V, \Sigma, R, S)$ where R is defined by including for each $A \in V$, all rules of the form $A \rightarrow \alpha$, where $B \rightarrow \alpha \in R$, $\alpha \notin V$ and $B \in \text{Chain}(A)$
Note: $A \in \text{Chain}(A)$ so all its non unit-rules are included

Non-Productive Symbols

- Let $G = (V, \Sigma, R, S)$ be an arbitrary CFG that has had its λ -rules and unit-rules removed
- Non-productive non-terminal symbols never lead to a terminal string (not productive)
- Productive(G) is computed by
Productive(G) $\supseteq \{ A \mid A \rightarrow \alpha, \alpha \in \Sigma^* \}$
Repeat
Productive(G) $\supseteq \{ B \mid B \rightarrow \alpha, \alpha \in (\Sigma \cup \text{Productive})^* \}$
until no new symbols are added
- Keep only those rules that involve productive symbols
- If no rules remain, grammar generates nothing

Unreachable Symbols

- Let $G = (V, \Sigma, R, S)$ be an arbitrary CFG that has had its λ -rules, unit-rules and non-productive symbols removed
- Unreachable symbols are ones that are inaccessible from start symbol
- We compute the complement (Useful)
- Useful(G) is computed by
Useful(G) \supseteq { S }
Repeat
 Useful(G) \supseteq { C | $B \rightarrow \alpha C \beta$, $C \in V \cup \Sigma$, $B \in \text{Useful}(G)$ }
until no new symbols are added
- Keep only those rules that involve useful symbols
- If no rules remain, grammar generates nothing

Reduced CFG

- A reduced CFG is one without λ -rules (except possibly for start symbol), no unit-rules, no non-productive symbols and no useless symbols

CFG to CNF

- Let $G = (V, \Sigma, R, S)$ be arbitrary reduced CFG
- Define $G' = (V \cup \{ \langle a \rangle \mid a \in \Sigma \}, \Sigma, R, S)$
- Add the rules $\langle a \rangle \rightarrow a$, for all $a \in \Sigma$
- For any rule, $A \rightarrow \alpha$, $|\alpha| > 1$, change each terminal symbol, a , in α to the non-terminal $\langle a \rangle$
- Now, for each rule $A \rightarrow BC\alpha$, $|\alpha| > 0$, introduce the new non-terminal $B\langle C\alpha \rangle$, and replace the rule $A \rightarrow BC\alpha$ with the rule $A \rightarrow B\langle C\alpha \rangle$ and add the rule $\langle C\alpha \rangle \rightarrow C\alpha$
- Iteratively apply the above step until all rules are in CNF

Example of CNF Conversion

Starting Grammars

- $L = \{ a^i b^j c^k \mid i=j \text{ or } j=k \}$
- $G = (\{S, A, \langle B=C \rangle, C, \langle A=B \rangle\}, \{a, b\}, R, S)$
- **R:**
 - $S \rightarrow A \mid C$
 - $A \rightarrow a A \mid \langle B=C \rangle$
 - $\langle B=C \rangle \rightarrow b \langle B=C \rangle c \mid \lambda$
 - $C \rightarrow C c \mid \langle A=B \rangle$
 - $\langle A=B \rangle \rightarrow a \langle A=B \rangle b \mid \lambda$

Remove Null Rules

- **Nullable = { $\langle B=C \rangle$, $\langle A=B \rangle$, A, C, S}**
 - $S' \rightarrow S \mid \lambda$ // S' added to V
 - $S \rightarrow A \mid C$
 - $A \rightarrow a A \mid a \mid \langle B=C \rangle$
 - $\langle B=C \rangle \rightarrow b \langle B=C \rangle c \mid b c$
 - $C \rightarrow C c \mid c \mid \langle A=B \rangle$
 - $\langle A=B \rangle \rightarrow a \langle A=B \rangle b \mid ab$

Remove Unit Rules

- Chains=

$\{[S':S',S,A,C,<A=B>,<B=C>],[S:S,A,C,<A=B>,<B=C>],$
 $[A:A,<B=C>],[C:C,<B=C>],[<B=C>:<B=C>],$
 $[<A=B>:<A=B>]\}$

- $S' \rightarrow \lambda \mid aA \mid a \mid b<B=C>c \mid bc \mid Cc \mid c \mid a<A=B>b \mid ab$
- $S \rightarrow aA \mid a \mid b<B=C>c \mid bc \mid Cc \mid c \mid a<A=B>b \mid ab$
- $A \rightarrow aA \mid a \mid b<B=C>c \mid bc$
- $<B=C> \rightarrow b<B=C>c \mid bc$
- $C \rightarrow Cc \mid c \mid a<A=B>b \mid ab$
- $<A=B> \rightarrow a<A=B>b \mid ab$

Remove Useless Symbols

- All non-terminal symbols are productive (lead to terminal string)
- S is useless as it is unreachable from S' (new start).
- All other symbols are reachable from S'

Normalize rhs as CNF

- $S' \rightarrow \lambda \mid \langle a \rangle A \mid a \mid \langle b \rangle \langle \langle B=C \rangle \langle c \rangle \rangle \mid \langle b \rangle \langle c \rangle \mid C \langle c \rangle \mid c \mid \langle a \rangle \langle \langle A=B \rangle \langle b \rangle \rangle \mid \langle a \rangle \langle b \rangle$
- $A \rightarrow \langle a \rangle A \mid a \mid \langle b \rangle \langle \langle B=C \rangle \langle c \rangle \rangle \mid \langle b \rangle \langle c \rangle$
- $\langle B=C \rangle \rightarrow \langle b \rangle \langle \langle B=C \rangle \langle c \rangle \rangle \mid \langle b \rangle \langle c \rangle$
- $C \rightarrow C \langle c \rangle \mid c \mid \langle a \rangle \langle \langle A=B \rangle \langle b \rangle \rangle \mid \langle a \rangle \langle b \rangle$
- $\langle A=B \rangle \rightarrow \langle a \rangle \langle \langle A=B \rangle \langle b \rangle \rangle \mid \langle a \rangle \langle b \rangle$
- $\langle \langle B=C \rangle \langle c \rangle \rangle \rightarrow \langle B=C \rangle \langle c \rangle$
- $\langle \langle A=B \rangle \langle b \rangle \rangle \rightarrow \langle A=B \rangle \langle b \rangle$
- $\langle a \rangle \rightarrow a$
- $\langle b \rangle \rightarrow b$
- $\langle c \rangle \rightarrow c$

CKY (Cocke, Kasami, Younger) $O(N^3)$ PARSING

Dynamic Programming

To solve a given problem, we solve small parts of the problem (subproblems), then combine the solutions of the subproblems to reach an overall solution.

The Parsing problem for arbitrary CFGs was elusive, in that its complexity was unknown until the late 1960s. In the meantime, theoreticians developed notion of simplified forms that were as powerful as arbitrary CFGs. The one most relevant here is the Chomsky Normal Form – CNF. It states that the only rule forms needed are:

$A \rightarrow BC$ where B and C are non-terminals

$A \rightarrow a$ where a is a terminal

This is provided the string of length zero is not part of the language.

CKY (Bottom-Up Technique)

Let the input string be a sequence of n letters $a_1 \dots a_n$.

Let the grammar contain r terminal and nonterminal symbols $R_1 \dots R_r$,

Let R_1 be the start symbol.

Let $P[n,n]$ be an array of Sets over $\{1, \dots, n\}$. Initialize all elements of P to empty ($\{\}$).

For each col = 1 to n

 For each unit production $X \rightarrow a_i$, set add X to $P[1, \text{col}]$.

For each row = 2 to n

 For each col = 1 to $n - \text{row} + 1$

 For each row2 = 1 to $\text{row} - 1$

 if $B \in P[\text{row2}, \text{col}]$ and $C \in P[\text{row} - \text{row2}, \text{col} + \text{row2}]$ and $A \rightarrow B C$ then

 add A to $P[\text{row}, \text{col}]$

If $R_1 \in P[n, n]$ is true then $a_1 \dots a_n$ is member of language

else $a_1 \dots a_n$ is not a member of language

CKY Parser

Present the **CKY** recognition matrix for the string **abba** assuming the Chomsky Normal Form grammar, $G = (\{S,A,B,C,D,E\}, \{a,b\}, R, S)$, specified by the rules **R**:

S → **AB | BA**
A → **CD | a**
B → **CE | b**
C → **a | b**
D → **AC**
E → **BC**

	a	b	b	a
1	A,C	B,C	B,C	A,C
2	S,D	E	S,E	
3	B	B		
4	S,E			

2nd CKY Example

E → **EF | ME | PE | a**
F → **MF | PF | ME | PE**
P → **+**
M → **-**

	a	-	a	+	a	-	a
1	E	M	E	P	E	M	E
2		E, F		E, F		E, F	
3	E		E		E		
4		E, F		E, F			
5	E		E				
6		E, F					
7	E						

Assignment # 6

1. Write a CFG for the following languages:
 $L = \{ a^p b^q c^r \mid p = |q - r| \}$ where $|x|$ is absolute value of x
2. Convert the following grammar to a CNF equivalent grammar. Show all steps.
 $G = (\{S, A, B\}, \{a, b\}, S, R)$ where R is:
 $S \rightarrow SS \mid ABA$
 $A \rightarrow ABB \mid a$
 $B \rightarrow BS \mid b \mid \lambda$
3. Present the **CKY** recognition matrix for the string $a - (b - a) + a$ assuming the Chomsky Normal Form grammar, $G = (\{E, F, G, H, K, LM, P, Q\}, \{a, b, +, -, (,)\}, E, R)$, where R is:
 $E \rightarrow EG \mid EH \mid LK \mid a \mid b$
 $G \rightarrow PF$
 $H \rightarrow MF$
 $K \rightarrow EQ$
 $F \rightarrow a \mid b \mid LK$
 $P \rightarrow +$
 $M \rightarrow -$
 $L \rightarrow ($
 $Q \rightarrow)$

Due: Tuesday, Oct. 8, 11:59 PM (use Webcourses to turn in)

Pumping Lemma for Context Free Languages

What is not a CFL

CFL Pumping Lemma

Concept

- Let L be a context free language then there is CNF grammar $G = (V, \Sigma, R, S)$ such that $L(G) = L$.
- As G is in CNF all its rules that allow the string to grow are of the form $A \rightarrow BC$, and thus growth has a binary nature.
- Any sufficiently long string z in L will have a parse tree that must have deep branches to accommodate z 's growth.
- Because of the binary nature of growth, the width of a tree with maximum branch length k at its deepest nodes is at most 2^k ; moreover, if the frontier of the tree is all terminal, then the string so produced is of length at most 2^{k-1} ; since the last rule applied for each leaf is of the form $A \rightarrow a$.
- Any terminal branch in a derivation tree of height $> |V|$ has more than $|V|$ internal nodes labelled with non-terminals. The “pigeonhole principle” tells us that whenever we visit $|V| + 1$ or more nodes, we must use at least one variable label more than once. This creates a self-embedding property that is key to the repetition patterns that occur in the derivation of sufficiently long strings.

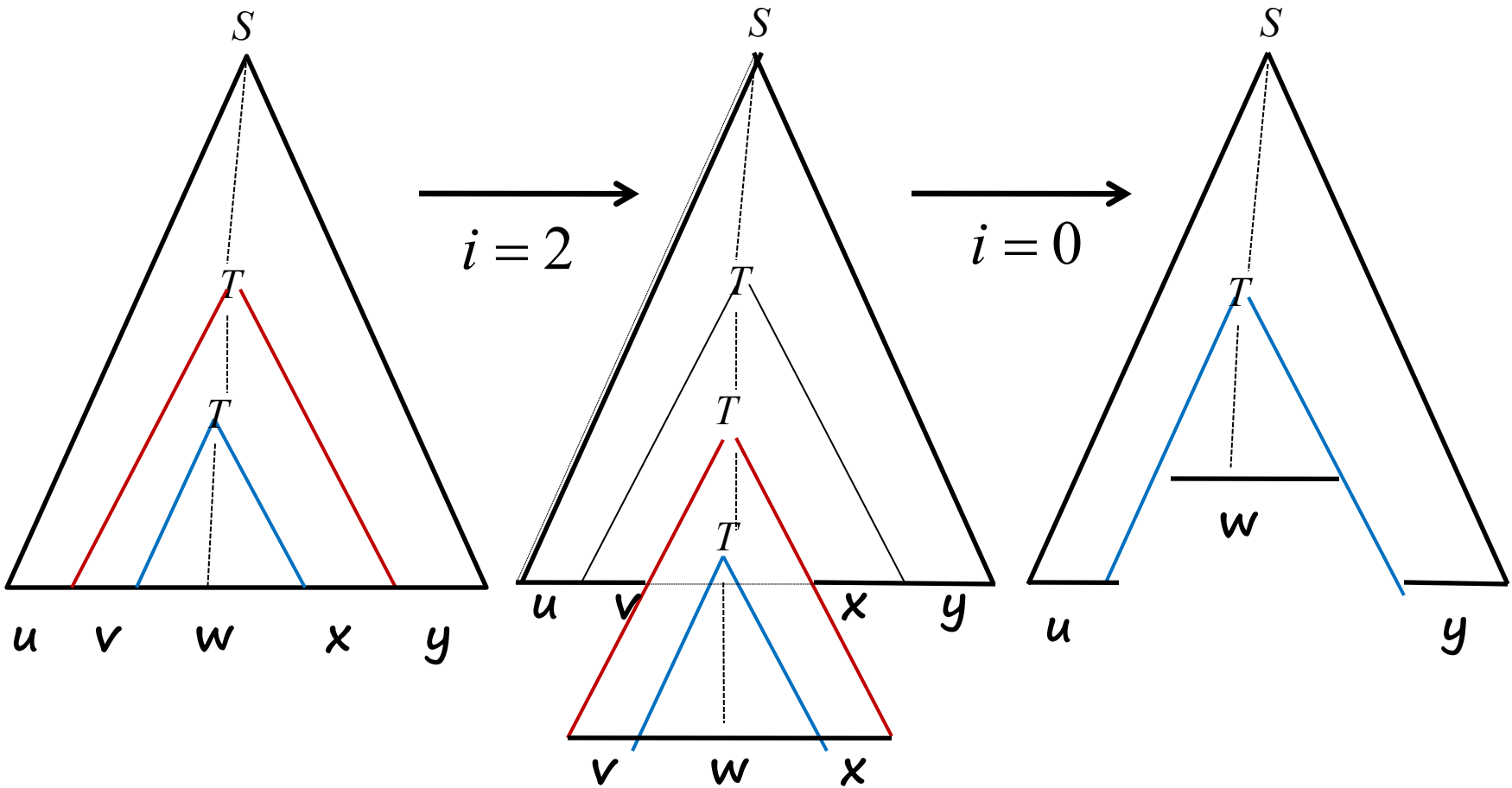
Pumping Lemma For CFL

- Let L be a CFL then there exists an $N > 0$ such that, if $z \in L$ and $|z| \geq N$, then z can be written in the form $uvwxy$, where $|vwx| \leq N$, $|vx| > 0$, and for all $i \geq 0$, $uv^iwx^iy \in L$.
- This means that interesting context free languages (infinite ones) have a self-embedding property that is symmetric around some central area, unlike regular where the repetition has no symmetry and occurs at the start.

Pumping Lemma Proof

- If L is a CFL then it is generated by some CNF grammar, $G = (V, \Sigma, R, S)$. Let $|V| = k$. For any string z , such that $|z| \geq N = 2^k$, the derivation tree for z based on G must have a branch with at least $k+1$ nodes labelled with variables from G .
- By the PigeonHole Principle at least two of these labels must be the same. Let the first repeated variable be T and consider the last two instances of T on this path.
- Let $z = uvwxy$, where $S \Rightarrow^* uTy \Rightarrow^* uvTxy \Rightarrow^* uvwxy$
- Clearly, then, we know $S \Rightarrow^* uTy$; $T \Rightarrow^* vTx$; and $T \Rightarrow^* w$
- But then, we can start with $S \Rightarrow^* uTy$; repeat $T \Rightarrow^* vTx$ zero or more times; and then apply $T \Rightarrow^* w$.
- But then, $S \Rightarrow^* uv^iwx^iy$ for all $i \geq 0$, and thus $uv^iwx^iy \in L$, for all $i \geq 0$.

Visual Support of Proof



Lemma's Adversarial Process

- Assume $L = \{a^n b^n c^n \mid n > 0\}$ is a CFL
- P.L.: Provides $N > 0$ We CANNOT choose N ; that's the P.L.'s job
- Our turn: Choose $a^N b^N c^N \in L$ We get to select a string in L
- P.L.: $a^N b^N c^N = uvwxy$, where $|vwx| \leq N$, $|vx| > 0$, and for all $i \geq 0$, $uv^i wx^i y \in L$ We CANNOT choose split, but P.L. is constrained by N
- Our turn: Choose $i=0$. We have the power here
- P.L: Two cases:
 - (1) vx contains some a 's and maybe some b 's. Because $|vwx| \leq N$, it cannot contain c 's if it has a 's. $i=0$ erases some a 's but we still have N c 's so $uwy \notin L$
 - (2) vx contains no a 's. Because $|vx| > 0$, vx contains some b 's or c 's or some of each. $i=0$ erases some b 's and/or c 's but we still have N a 's so $uwy \notin L$
- CONTRADICTION, so L is NOT a CFL

Non-Closure

- Intersection ($\{ a^n b^n c^n \mid n \geq 0 \}$ is not a CFL)
 $\{ a^n b^n c^n \mid n \geq 0 \} =$
 $\{ a^n b^n c^m \mid n, m \geq 0 \} \cap \{ a^m b^n c^n \mid n, m \geq 0 \}$
Both of the above are CFLs
- Complement
If closed under complement then would be closed under Intersection as
 $A \cap B = \sim(\sim A \cup \sim B)$

Max and Min of CFL

- Consider the two operations on languages max and min, where
 - $\max(L) = \{ x \mid x \in L \text{ and, for no non-null } y \text{ does } xy \in L \}$ and
 - $\min(L) = \{ x \mid x \in L \text{ and, for no proper prefix of } x, y, \text{ does } y \in L \}$
- Describe the languages produced by max and min. for each of :
 - $L1 = \{ a^i b^j c^k \mid k \leq i \text{ or } k \leq j \}$ CFL
 - $\max(L1) = \{ a^i b^j c^k \mid k = \max(i, j) \}$ Non-CFL
 - $\min(L1) = \{ \lambda \}$ (string of length 0) Regular
 - $L2 = \{ a^i b^j c^k \mid k > i \text{ or } k > j \}$ CFL
 - $\max(L2) = \{ \}$ (empty) Regular
 - $\min(L2) = \{ a^i b^j c^k \mid k = \min(i, j) + 1 \}$ Non-CFL
- $\max(L1)$ shows CFL not closed under max
- $\min(L2)$ shows CFL not closed under min

Complement of ww

- Let $L = \{ ww \mid w \in \{a,b\}^+ \}$. L is not a CFL
- Consider L 's complement, it must be of form $xayx'by'$ or $xbyx'ay'$, where $|x|=|x'|$ and $|y|=|y'|$
- The above reflects that this language has one “transcription error”
- This seems really hard to write a CFG but it's all a matter of how you view it
- We don't care about what precedes or follows the errors so long as the lengths are right
- Thus, we can view above as $xax'yby'$ or $xbx'y'ay'$, where $|x|=|x'|$ and $|y|=|y'|$
- The grammar for this has rules
 $S \rightarrow AB \mid BA$; $A \rightarrow XAX \mid a$; $B \rightarrow XBX \mid b$
 $X \rightarrow a \mid b$

Solvable CFL Problems

- Let L be an arbitrary CFL generated by CFG G with start symbol S then the following are all decidable
 - Is w in L ?
Run CKY
If S in final cell then $w \in L$
 - Is L empty (non-empty)?
Reduce G
If no rules left then empty
 - Is L finite (infinite)?
Reduce G
Run DFS(S)
If no loops then finite

Assignment # 7

1. Use the Pumping Lemma for CFLs to prove that none of the following are CFLs.
 - a) $L = \{ a^i b^j c^k d^m \mid m = \min(\max(i,j), k) \}$
 - b) $L = \{ a^i b^j \mid j = \sum_{k=1}^i k \}$
 - c) $L = \{ w w^R w \mid w \in \{a,b\}^+ \}$

Due: Thursday Oct. 10, 11:59PM (use Webcourses to turn in)

Closure Properties

Context Free Languages

Intersection with Regular

- CFLs are closed under intersection with Regular sets
 - To show this we use the equivalence of CFGs generative power with the recognition power of PDAs (shown later).
 - Let $A_0 = (Q_0, \Sigma, \Gamma, \delta_0, q_0, \$, F_0)$ be an arbitrary PDA
 - Let $A_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ be an arbitrary DFA
 - Define $A_2 = (Q_0 \times Q_1, \Sigma, \Gamma, \delta_2, \langle q_0, q_1 \rangle, \$, F_0 \times F_1)$ where
 - $\delta_2(\langle q, s \rangle, a, X) \supseteq \{(\langle q', s' \rangle, \alpha)\}$, $a \in \Sigma \cup \{\lambda\}$, $X \in \Gamma$ iff
 - $\delta_0(q, a, X) \supseteq \{(q', \alpha)\}$ and
 - $\delta_1(s, a) = s'$ (if $a = \lambda$ then $s' = s$).
 - Using the definition of derivations we see that
 - $[\langle q_0, q_1 \rangle, w, \$] \vdash^* [\langle t, s \rangle, \lambda, \beta]$ in A_2 iff
 - $[q_0, w, \$] \vdash^* [t, \lambda, \beta]$ in A_0 and
 - $[q_1, w] \vdash^* [s, \lambda]$ in A_1
- But then $w \in \mathcal{F}(A_2)$ iff $t \in F_0$ and $s \in F_1$ iff $w \in \mathcal{F}(A_0)$ and $w \in \mathcal{F}(A_1)$

Substitution

- CFLs are closed under CFL substitution
 - Let $G=(V,\Sigma,R,S)$ be a CFG.
 - Let f be a substitution over Σ such that
 - $f(a) = L_a$ for $a \in \Sigma$
 - $G_a = (V_a,\Sigma_a,R_a,S_a)$ is a CFG that produces L_a .
 - No symbol appears in more than one of V or any V_a
 - Define $G_f = (V \cup_{a \in \Sigma} V_a, \cup_{a \in \Sigma} \Sigma_a, R' \cup_{a \in \Sigma} R_a, S)$
 - $R' = \{ A \rightarrow g(\alpha) \text{ where } A \rightarrow \alpha \text{ is in } R \}$
 - $g: (V \cup \Sigma)^* \rightarrow (V \cup_{a \in \Sigma} S_a)^*$
 - $g(\lambda) = \lambda; g(B) = B, B \in V; g(a) = S_a, a \in \Sigma$
 - $g(\alpha X) = g(\alpha) g(X), |\alpha| > 0, X \in V \cup \Sigma$
 - Claim, $f(\mathcal{L}(G)) = \mathcal{L}(G_f)$, and so CFLs closed under substitution and homomorphism.

More on Substitution

- Consider G'_f . If we limit derivations to the rules $R' = \{ A \rightarrow g(\alpha) \text{ where } A \rightarrow \alpha \text{ is in } R \}$ and consider only sentential forms over the $\cup_{a \in \Sigma} S_a$, then $S \Rightarrow^* S_{a_1} S_{a_2} \dots S_{a_n}$ in G' iff $S \Rightarrow^* a_1 a_2 \dots a_n$ iff $a_1 a_2 \dots a_n \in \mathcal{L}(G)$. But, then $w \in \mathcal{L}(G)$ iff $f(w) \in \mathcal{L}(G_f)$ and, thus, $f(\mathcal{L}(G)) = \mathcal{L}(G_f)$.
- Given that CFLs are closed under intersection, substitution, homomorphism and intersection with regular sets, we can recast previous proofs to show that CFLs are closed under
 - Prefix, Suffix, Substring, Quotient with Regular Sets
- Later we will show that CFLs are not closed under Quotient with CFLs.

Midterm Topics.1

- Finite-state automata and Regular languages
 - Definitions: Deterministic and Non-Deterministic
 - Notions of state transitions, acceptance and language accepted
 - State diagrams and state tables
 - Construction from descriptions of languages
 - Conversion of NFA to DFA
 - λ -Closure
 - Subset construction
 - Reachable states
 - Reaching states
 - Minimizing DFAs (distinguishable states)

Midterm Topics.2

- Regular expressions and Regular Sets
 - Definition of regular expressions and regular sets
 - Every regular sets is a regular language
 - Every regular language is a regular set
 - Ripping states (GNFA)
 - $R_{i,j}^k$ expressions
 - $R_{ij}^{k+1} = R_{ij}^k + (R_{i(k+1)}^k \cdot (R_{(k+1)(k+1)}^k)^* \cdot R_{(k+1)j}^k)$
 - $L(A) = \bigcup_{f \in F} R_{1f}^n$
 - Regular equations
 - Uniqueness of solution to $R=Q+RP$
 - Solving for expressions associated with states

Midterm Topics.3

- Pumping Lemma
 - Classic non-regular languages $\{0^n 1^n \mid n \geq 0\}$
 - Formal statement and proof of Pumping Lemma for Regular Languages
 - Use of Pumping Lemma
- Minimization (using distinguishable states)
- Myhill-Nerode
 - Right Invariant Equivalence Relations (RIER)
 - Specific RIER, $x R_L y \forall z [xz \in L \Leftrightarrow yz \in L]$ is minimal
 - Uniqueness of minimum state DFA based on R_L
 - Use to show languages are no Regular

Midterm Topics.4

- Grammars
 - Definition of grammar and notions of derivation and language
 - Restricted grammars: Regular (right and left linear)
 - Why you can't mix right and left linear and stay in Regular domain
 - Relation of regular grammars to finite-state automata
- Closures
 - Union, Concatenation, Keene star
 - Complement, Exclusive Union, Intersection, Set Difference, Reversal
 - Substitution, Homomorphism, Quotient, Prefix, Suffix, Substring
 - Max, Min
- Decidable Properties
 - Membership
 - $L = \emptyset$
 - $L = \Sigma^*$
 - Finiteness / Infiniteness
 - Equivalence

Midterm Topics.5

- Context free grammars
 - Writing grammars for specific languages
 - Leftmost and rightmost derivations, Parse trees, Ambiguity
 - Closure (union, concatenation, reversal, substitution, homomorphism)
 - Pumping Lemma for CFLs
 - Chomsky Normal Form
 - Remove lambda rules
 - Remove chain rules
 - Remove non-generating (unproductive) non-terminals (and rules)
 - Remove unreachable non-terminals (and rules)
 - Make rhs match CNF constraints
 - CKY algorithm

Midterm Topics.6

- Closure
 - Union, concatenation, star
 - Substitution
 - Intersection with regular
 - Quotient with regular, Prefix, Suffix, Substring
- Non-Closure
 - Intersection, complement, min, max

Push Down Automata

CFL Recognizers

Formalization of PDA

- $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$
- Q is finite set of states
- Σ is finite input alphabet
- Γ is finite set of stack symbols
- $\delta : Q \times \Sigma_e \times \Gamma_e \rightarrow 2^{Q \times \Gamma^*}$ is transition function
 - Note: Can limit stack push to Γ_e but it's equivalent!!
- $Z_0 \in \Gamma$ is an optional initial symbol on stack
- $F \subseteq Q$ is final set of states and can be omitted for some notions of a PDA

Notion of ID for PDA

- An instantaneous description for a PDA is $[q, w, \gamma]$ where
 - q is current state
 - w is remaining input
 - γ is contents of stack (leftmost symbol is top)
- Single step derivation is defined by
 - $[q, ax, Z\alpha] \vdash [p, x, \beta\alpha]$ if $\delta(q, a, Z)$ contains (p, β)
- Multistep derivation (\vdash^*) is the reflexive transitive closure of single step.

Language Recognized by PDA

- Given $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$
there are three senses of recognition
- By final state
 $L(A) = \{w | [q_0, w, Z_0] \vdash^* [f, \lambda, \beta]\}$, where $f \in F$
- By empty stack
 $N(A) = \{w | [q_0, w, Z_0] \vdash^* [q, \lambda, \lambda]\}$
- By empty stack and final state
 $E(A) = \{w | [q_0, w, Z_0] \vdash^* [f, \lambda, \lambda]\}$, where $f \in F$

Top Down Parsing by PDA

- Given $G = (V, \Sigma, R, S)$, define
 $A = (\{q\}, \Sigma, \Sigma \cup V, \delta, q, S, \phi)$
- $\delta(q, a, a) = \{(q, \lambda)\}$ for all $a \in \Sigma$
- $\delta(q, \lambda, A) = \{(q, \alpha) \mid A \rightarrow \alpha \in R \text{ (guess)}\}$
- $N(A) = L(G)$

- Has just one state, so is essentially stateless, except for stack content

Top Down Parsing by PDA

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid \text{Int}$

• $\delta(q, +, +) = \{(q, \lambda)\}$, $\delta(q, *, *) = \{(q, \lambda)\}$,

• $\delta(q, \text{Int}, \text{Int}) = \{(q, \lambda)\}$,

• $\delta(q, (, () = \{(q, \lambda)\}$, $\delta(q,),) = \{(q, \lambda)\}$

• $\delta(q, \lambda, E) = \{(q, E+T), (q, T)\}$

• $\delta(q, \lambda, T) = \{(q, T*F), (q, F)\}$

• $\delta(q, \lambda, F) = \{(q, (E)), (q, \text{Int})\}$

Bottom Up Parsing by PDA

- Given $G = (V, \Sigma, R, S)$, define
 $A = (\{q, f\}, \Sigma, \Sigma \cup V \cup \{\$, \delta, q, \$, \{f\})$
- $\delta(q, a, \lambda) = \{(q, a)\}$ for all $a \in \Sigma$, SHIFT
- $\delta(q, \lambda, \alpha^R) \supseteq \{(q, A)\}$ if $A \rightarrow \alpha \in R$, REDUCE
Cheat: looking at more than top of stack
- $\delta(q, \lambda, S) \supseteq \{(f, \lambda)\}$
- $\delta(f, \lambda, \$) = \{(f, \lambda)\}$, ACCEPT
- $E(A) = L(G)$
- Could also do $\delta(q, \lambda, S\$) \supseteq \{(q, \lambda)\}$, $N(A) = L(G)$

Bottom Up Parsing by PDA

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid \text{Int}$

• $\delta(q, +, \lambda) = \{(q, +)\}$, $\delta(q, *, \lambda) = \{(q, *)\}$, $\delta(q, \text{Int}, \lambda) = \{(q, \text{Int})\}$,

$\delta(q, (, \lambda) = \{(q, ()\}$, $\delta(q,), \lambda) = \{(q,)\}$

• $\delta(q, \lambda, T + E) = \{(q, E)\}$, $\delta(q, \lambda, T) \supseteq \{(q, E)\}$

• $\delta(q, \lambda, F * T) \supseteq \{(q, T)\}$, $\delta(q, \lambda, F) \supseteq \{(q, T)\}$

• $\delta(q, \lambda,)E() \supseteq \{(q, F)\}$, $\delta(q, \lambda, \text{Int}) \supseteq \{(q, F)\}$

• $\delta(q, \lambda, E) \supseteq \{(f, \lambda)\}$

• $\delta(f, \lambda, \$) = \{(f, \lambda)\}$

• $E(A) = L(G)$

Challenge

- Use the two recognizers on some sets of expressions like

$$- 5 + 7 * 2$$

$$- 5 * 7 + 2$$

$$- (5 + 7) * 2$$

Converting a PDA to CFG

- Book has one approach; here is another
- Let $A = (Q, \Sigma, \Gamma, \delta, q_0, Z, F)$ accept L by empty stack and final state
- Define $A' = (Q \cup \{q_0', f\}, \Sigma, \Gamma \cup \{\$\}, \delta', q_0', \$, \{f\})$ where
 - $\delta'(q_0', \lambda, \$) = \{(q_0, \text{PUSH}(Z))\}$ or in normal notation $\{(q_0, Z\$)\}$
 - δ' does what δ does but only uses PUSH and POP instructions, always reading top of stack
Note1: we need to consider using the $\$$ for cases of the original machine looking at empty stack, when using λ for stack check. This guarantees we have top of stack until very end.
Note2: If original adds stuff to stack, we do pop, followed by a bunch of pushes.
 - We add $(f, \lambda) = (f, \text{POP})$ to $\delta'(q_f, \lambda, \$)$ whenever q_f is in F , so we jump to a fixed final state.
- Now, wlog, we can assume our PDA uses only POP and PUSH, has just one final state and accepts by empty stack and final state. We will assume the original machine is of this form and that its bottom of stack is $\$$.
- Define $G = (V, \Sigma, R, S)$ where
 - $V = \{S\} \cup \{ \langle q, X, p \rangle \mid q, p \in Q, X \in \Gamma \}$
 - R on next page

Rules for PDA to CFG

- R contains rules as follows:
 $S \rightarrow \langle q_0, \$, f \rangle$ where $F = \{f\}$
meaning that we want to generate w whenever
 $[q_0, w, \$] \vdash^* [f, \lambda, \lambda]$
- Remaining rules are:
 $\langle q, X, p \rangle \rightarrow a \langle s, Y, t \rangle \langle t, X, p \rangle$
whenever $\delta(q, a, X) \ni \{(s, \text{PUSH}(Y))\}$
 $\langle q, X, p \rangle \rightarrow a$
whenever $\delta(q, a, X) \ni \{(p, \text{POP})\}$
- Want $\langle q, X, p \rangle \Rightarrow^* w$ when $[q, w, X] \vdash^* [p, \lambda, \lambda]$

Context Sensitive

Context Sensitive Grammar

$G = (V, \Sigma, R, S)$ is a PSG where

Each member of R is a rule whose right side is no shorter than its left side.

The essential idea is that rules are length preserving, although we do allow $S \rightarrow \lambda$ so long as S never appears on the right hand side of any rule.

A context sensitive grammar is denoted as a CSG and the language generated is a Context Sensitive Language (CSL).

The recognizer for a CSL is a Linear Bounded Automaton (LBA), a form of Turing Machine (soon to be discussed), but with the constraint that it is limited to moving along a tape that contains just the input surrounded by a start and end symbol.

Phrase Structured Grammar

We previously defined PSGs. The language generated by a PSG is a Phrase Structured Language (PSL) but is more commonly called a recursively enumerable (re) language. The reason for this will become evident a bit later in the course.

The recognizer for a PSL (re language) is a Turing Machine, a model of computation we will soon discuss.

CSG Example#1

$$L = \{ a^n b^n c^n \mid n > 0 \}$$

$G = (\{A, B, C\}, \{a, b, c\}, R, A)$ where R is

$$A \rightarrow aBbc \mid abc$$

$$B \rightarrow aBbC \mid abC$$

Note: $A \Rightarrow aBbc \Rightarrow_n a^{n+1}(bC)^n bc \quad // \ n > 0$

$Cb \rightarrow bC \quad // \text{ Shuttle } C \text{ over to a } c$

$Cc \rightarrow cc \quad // \text{ Change } C \text{ to a } c$

Note: $a^{n+1}(bC)^n bc \Rightarrow^* a^{n+1}b^{n+1}c^{n+1}$

Thus, $A \Rightarrow^* a^n b^n c^n, n > 0$

CSG Example#2

$L = \{ ww \mid w \in \{0,1\}^+ \}$

$G = (\{S,A,X,Z,<0>,<1>\}, \{0,1\}, R, S)$ where R is

$S \rightarrow 00 \mid 11 \mid 0A<0> \mid 1A<1>$

$A \rightarrow 0AZ \mid 1AX \mid 0Z \mid 1X$

$Z0 \rightarrow 0Z$ $Z1 \rightarrow 1Z$ // Shuttle Z (for owe zero)

$X0 \rightarrow 0X$ $X1 \rightarrow 1X$ // Shuttle X (for owe one)

$Z<0> \rightarrow 0<0>$ $Z<1> \rightarrow 1<0>$ // New 0 must be on rhs of old 0/1's

$X<0> \rightarrow 0<1>$ $X<1> \rightarrow 1<1>$ // New 1 must be on rhs of old 0/1's

$<0> \rightarrow 0$ // Guess we are done

$<1> \rightarrow 1$ // Guess we are done

Topics Deferred to Final

- Push-down automata
 - Various notions of acceptance and their equivalence
 - Deterministic vs non-deterministic
 - Equivalence to CFLs
 - CFG to PDA definitely; **PDA to CFG, only conceptually**
 - Top-down vs bottom up parsing via PDAs
- Context sensitive grammars and LBAs
 - Rules for CSG
 - Ability to shuttle symbols to preset places
 - Just basic definition of LBA

Computability

The study of models of computation and what can/cannot be done via purely mechanical means

Goals of Computability

- Provide precise characterizations (computational models) of the class of effective procedures / algorithms.
- Study the boundaries between complete and incomplete models of computation.
- Study the properties of classes of solvable and unsolvable problems.
- Solve or prove unsolvable open problems.
- Determine reducibility and equivalence relations among unsolvable problems.
- Our added goal is to apply these techniques and results across multiple areas of Computer Science.

Hilbert, Russell and Whitehead

- Late 1800's to early 1900's
- Russell and Whitehead: Principia Mathematica
 - Developed and catalogued axiomatic schemes
 - Axioms plus sound rules of inference
 - Much of focus on number theory
- Hilbert
 - Felt all mathematics could be developed within a formal system that allowed the mechanical creation and checking of proofs
 - Even posed 23 problems, the solutions to which he felt were critical to understanding how to attack hard problems
- Post
 - Devised truth tables as an algorithmic approach to checking Boolean propositions for tautologies and satisfiability

Gödel

- In 1931 Gödel showed that any first order theory that embeds elementary arithmetic is either incomplete or inconsistent.
- Gödel also developed the general notion of recursive functions but made no claims about their strength.
 - We will look at the formal description of recursive functions later

Turing (Post, Church, Kleene)

- In 1936, each presented a formalism for computability.
 - Turing and Post devised abstract machines and claimed these represented all mechanically computable functions.
 - Church developed the notion of lambda-computability from recursive functions (as previously defined by Gödel and Kleene) and claimed completeness for this model. Lambda calculus gave birth to Lisp.
- Kleene demonstrated the computational equivalence of recursively defined functions to Post-Turing machines.
- Post later showed computability could also be described by forms of symbolic rewriting systems.

Basic Definitions

The Preliminaries

Goals of Computability

- Provide precise characterizations (computational models) of the class of effective procedures / algorithms.
- Study the boundaries between complete and incomplete models of computation.
- Study the properties of classes of solvable and unsolvable problems.
- Solve or prove unsolvable open problems.
- Determine reducibility and equivalence relations among unsolvable problems.
- Our added goal is to apply these techniques and results across multiple areas of Computer Science.

Effective Procedure

- *A process whose execution is clearly specified to the smallest detail*
- Such procedures have, among other properties, the following:
 - Processes must be finitely describable and the language used to describe them must be over a finite alphabet.
 - The current state of the machine model must be finitely presentable.
 - Given the current state, the choice of actions (steps) to move to the next state must be easily determinable from the procedure's description.
 - Each action (step) of the process must be capable of being carried out in a finite amount of time.
 - The semantics associated with each step must be clear and unambiguous.

Algorithm

- *An effective procedure that halts on all input*
- The key term here is “*halts on all input*”
- By contrast, an effective procedure may halt on all, none or some of its input.
- The domain of an algorithm is its entire universe of possible inputs.
- *Useful Notations*
 - $f(x)\downarrow$ means procedure f converges/halts/produces an output, when evaluated at x .
 - $f(x)\uparrow$ means procedure f diverges, when evaluated at x .
 - f is an algorithm iff $\forall x f(x)\downarrow$

Sets and Decision Problems

- Set -- A collection of atoms from some universe U . \emptyset denotes the empty set.
- (Decision) Problem -- A set of questions about elements of some universe. Each question has answer “yes” or “no”. The elements having answer “yes” constitute a set that is a subset of the corresponding universe. Those having answer “no” constitute the complement of the “yes” set.

Categorizing Problems (Sets)

- Solvable or Decidable -- A problem P is said to be solvable (decidable) if there exists an algorithm F which, when applied to a question q in P , produces the correct answer (“yes” or “no”). This is an inherent property of P .
- Solved -- A problem P is said to be solved if P is solvable and we have produced its solution. This is a temporal property in that P may have been unsolved for many years before being solved.
- Unsolved, Unsolvable (Undecidable) --
Complements of above

Categorizing Problems (Sets) # 2

- Recursively enumerable -- A set S is recursively enumerable (re) if S is empty ($S = \emptyset$) or there exists an algorithm F , over the natural numbers \mathbf{N} , whose range is exactly S . A problem is said to be re if the set associated with it is re.
- Semi-Decidable -- A problem is said to be semi-decidable if there is an effective procedure F which, when applied to a question q in P , produces the answer “yes” if and only if q has answer “yes”. F need not halt if q has answer “no”.
- Semi-decidable is the same as the notion of recognizable used in the text.

Immediate Implications

- **P** solved implies **P** solvable implies **P** semi-decidable (re, recognizable).
- **P** non-re implies **P** unsolvable implies **P** unsolved.
- **P** finite implies **P** solvable.

Slightly Harder Implications

- \mathbf{P} enumerable iff \mathbf{P} semi-decidable.
- \mathbf{P} solvable iff both \mathbf{S}_P and $(\mathbf{U} - \mathbf{S}_P)$ are re (semi-decidable).

- We will prove these later.

Existence of Undecidables

- A counting argument
 - The number of mappings from N to N is at least as great as the number of subsets of N . But the number of subsets of N is uncountably infinite (\aleph_1). However, the number of programs in any model of computation is countably infinite (\aleph_0). This latter statement is a consequence of the fact that the descriptions must be finite and they must be written in a language with a finite alphabet. In fact, not only is the number of programs countable, it is also effectively enumerable; moreover, its membership is decidable.
- A diagonalization argument
 - Will be shown later in class

Hilbert's Tenth

Diophantine Equations are
Unsolvable

One Variable Diophantine
Equations are Solvable

Hilbert's 10th

- In 1900 declared there were 23 really important problems in mathematics.
- Belief was that the solutions to these would help address math's complexity.
- Hilbert's Tenth asks for an algorithm to find the integral roots of polynomials with integral coefficients. For example
 $6x^3yz^2 + 3xy^2 - x^3 - 10 = 0$ has roots
 $x = 5; y = 3; z = 0$
- This is now known to be impossible to solve (In 1970, Matiyacevič showed this undecidable).

Hilbert's 10th is Semi-Decidable

- Consider over one variable: $P(x) = 0$
- Can semi-decide by plugging in $0, 1, -1, 2, -2, 3, -3, \dots$
- This terminates and says “yes” if $P(x)$ evaluates to 0, eventually. Unfortunately, it never terminates if there is no x such that $P(x) = 0$.
- Can easily extend to $P(x_1, x_2, \dots, x_k) = 0$.

Turing Machines

1st Model

A Linear Memory Machine

Textbook Description

- A Turing machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$
- Q is finite set of states
- Σ , is a finite input alphabet not containing the blank symbol \sqcup
- Γ is finite set of tape symbols that includes Σ and \sqcup . Commonly $\Gamma = \Sigma \cup \{\sqcup\}$
- $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{R, L\}$
 - Each instance of $Q \times \Gamma$ is called a discriminant
- q_0 starts, q_{accept} accepts, q_{reject} rejects

Turing versus Post

- The Turing description just given requires you to write a new symbol and move off the current tape square at every step
- Post had a variant where
$$\delta: Q \times \Gamma \rightarrow (Q \times (\Gamma \cup \{R, L\})) \cup \emptyset$$
- Here, you either write or move, not both (or just flat stop)
- Also, Post did not have an accept or reject state – acceptance is giving an answer of 1; rejection is 0; this treats decision procedures as predicates (functions that map input into $\{0, 1\}$)
- The way we stop our machines from running is to omit actions for some discriminants making the transition function partial
- I tend to use Post's notation and to define macros so machines are easy to create
- I am not a fan of having you build Turing tables

Basic Description

- We will use a simplified form that is a variant of Post's models.
- Here, each machine is represented by a finite set of states Q , the simple alphabet $\{0,1\}$, where 0 is the blank symbol, and each state transition is defined by a 4-tuple of form

$q a X s$

where $q a$ is the discriminant based on current state q , scanned symbol a ; X can be one of $\{R, L, 0, 1\}$, signifying move right, move left, print 0, or 1; and s is the new state.

- Limiting the alphabet to $\{0,1\}$ is not really a limitation. We can represent a k -letter alphabet by encoding the j -th letter via j 1's in succession. A 0 ends each letter, and two 0's ends a word.
- We rarely write quads. Rather, we typically will build machines from simple forms.

Base Machines

- R -- move right over any scanned symbol
- L -- move left over any scanned symbol
- 0 -- write a 0 in current scanned square
- 1 -- write a 1 in current scanned square
- We can then string these machines together with optionally labeled arcs.
- A labeled arc signifies a transition from one part of the composite machine to another, if the scanned square's content matches the label. Unlabeled arcs are unconditional. We will put machines together without arcs, when the arcs are unlabeled.

Useful Composite Machines

\mathcal{R} -- move right to next 0 (not including current square)

...?11...10... \Rightarrow ...?11...10...



\mathcal{L} -- move left to next 0 (not including current square)

...011...1?... \Rightarrow ...011...1?...



Commentary on Machines

- These machines can be used to move over encodings of letters or encodings of unary based natural numbers.
- In fact, any effective computation can easily be viewed as being over natural numbers. We can get the negative integers by pairing two natural numbers. The first is the sign (0 for +, 1 for -). The second is the magnitude.

Computing with TMs

A reasonably standard definition of a Turing computation of some n -ary function F is to assume that the machine starts with a tape containing the n inputs, x_1, \dots, x_n in the form

$$\dots 01^{x_1} 01^{x_2} 0 \dots 01^{x_n} \underline{0} \dots$$

and ends with

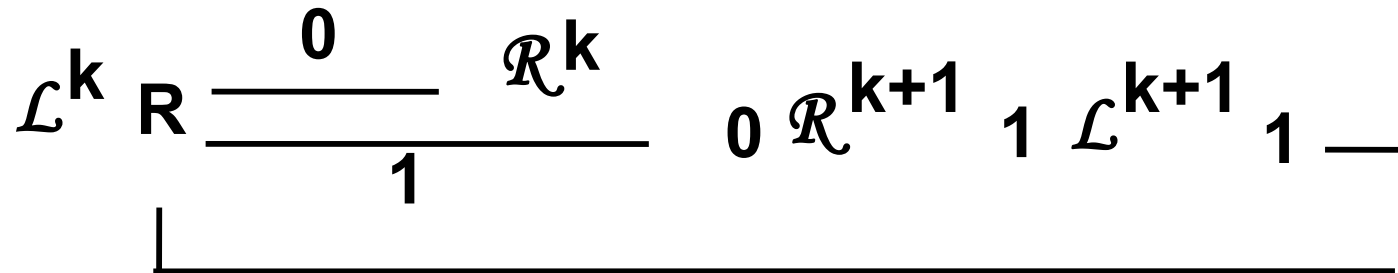
$$\dots 01^{x_1} 01^{x_2} 0 \dots 01^{x_n} 01^y \underline{0} \dots$$

where $y = F(x_1, \dots, x_n)$.

If we limit movement to never go left of the zero to the left of 1^{x_1} , we call this Standard Turing Computing (STC).

Addition by TM

Need the copy family of useful submachines, where C_k copies k -th preceding value.



The add machine is then

$$C_2 C_2 L 1 R L 0$$

Turing Machine Variations

- Two tracks
- N tracks
- Non-deterministic *****
- Two-dimensional
- K-dimensional
- Two stack machines
- Two counter machines

Register Machines

2nd Model

Feels Like Assembly Language

Register Machine Concepts

- A register machine consists of a finite length program, each of whose instructions is chosen from a small repertoire of simple commands.
- The instructions are labeled from **1** to **m**, where there are **m** instructions. Termination occurs as a result of an attempt to execute the **m+1**-st instruction.
- The storage medium of a register machine is a finite set of registers, each capable of storing an arbitrary natural number.
- Any given register machine has a finite, predetermined number of registers, independent of its input.

Computing by Register Machines

- A register machine partially computing some n -ary function F typically starts with its argument values in the first n registers and ends with the result in the $n+1$ -st register.
- We extend this slightly to allow the computation to start with values in its $k+1$ -st through $k+n$ -th register, with the result appearing in the $k+n+1$ -th register, for any k , such that there are at least $k+n+1$ registers.
- Sometimes, we use the notation of finishing with the results in the first register, and the arguments appearing in 2 to $n+1$.

Register Instructions

- Each instruction of a register machine is of one of two forms:

INC_r[i] –

increment **r** and jump to **i**.

DEC_r[p, z] –

if register **r** > **0**, decrement **r** and jump to **p**

else jump to **z**

Addition by RM

Addition ($r3 \leftarrow r1 + r2$)

1. DEC3[1,2] : Zero result (r3) and work (r4) registers
2. DEC4[2,3]
3. DEC1[4,6] : Add r1 to r3, saving original r1 in r4
4. INC3[5]
5. INC4[3]
6. DEC4[7,8] : Restore r1
7. INC1[6]
8. DEC2[9,11] : Add r2 to r3, saving original r2 in r4
9. INC3[10]
10. INC4[8]
11. DEC4[12,13] : Restore r2
12. INC2[11]
13. : Halt by branching here

Limited Subtraction by RM

Subtraction ($r3 \leftarrow r1 - r2$, if $r1 \geq r2$; 0, otherwise)

1. DEC3[1,2] : Zero result (r3) and work (r4) registers
2. DEC4[2,3]
3. DEC1[4,6] : Add r1 to r3, saving original r1 in r4
4. INC3[5]
5. INC4[3]
6. DEC4[7,8] : Restore r1
7. INC1[6]
8. DEC2[9,11] : Subtract r2 from r3, saving original r2 in r4
9. DEC3[10,10] : Note that decrementing 0 does nothing
10. INC4[8]
11. DEC4[12,13] : Restore r2
12. INC2[11]
13. : Halt by branching here

Factor Replacement Systems

3rd Model

Deceptively Simple

Factor Replacement Concepts

- A factor replacement system (FRS) consists of a finite (ordered) sequence of fractions, and some starting natural number \mathbf{x} .
- A fraction $\mathbf{a/b}$ is applicable to some natural number \mathbf{x} , just in case \mathbf{x} is divisible by \mathbf{b} . We always chose the first applicable fraction ($\mathbf{a/b}$), multiplying it times \mathbf{x} to produce a new natural number $\mathbf{x*a/b}$. The process is then applied to this new number.
- Termination occurs when no fraction is applicable.
- A factor replacement system partially computing \mathbf{n} -ary function \mathbf{F} typically starts with its argument encoded as powers of the first \mathbf{n} odd primes. Thus, arguments $\mathbf{x_1, x_2, \dots, x_n}$ are encoded as $\mathbf{3^{x_1} 5^{x_2} \dots p_n^{x_n}}$. The result then appears as the power of the prime $\mathbf{2}$.

Addition by FRS

Addition is $3^x 5^{x^2}$ becomes $2^{x^1+x^2}$

or, in more details, $2^0 3^{x^1} 5^{x^2}$ becomes $2^{x^1+x^2} 3^0 5^0$

$$2 / 3$$

$$2 / 5$$

Note that these systems are sometimes presented as rewriting rules of the form

$$\mathbf{bx} \rightarrow \mathbf{ax}$$

meaning that a number that has can be factored as \mathbf{bx} can have the factor \mathbf{b} replaced by an \mathbf{a} .

The previous rules would then be written

$$3x \rightarrow 2x$$

$$5x \rightarrow 2x$$

Limited Subtraction by FRS

Subtraction is $3^{x_1}5^{x_2}$ becomes $2^{\max(0, x_1-x_2)}$

$$3 \cdot 5x \rightarrow x$$

$$3x \rightarrow 2x$$

$$5x \rightarrow x$$

Challenge: How would you do $3^{x_1}5^{x_2}$ becomes $2^{|x_1-x_2|}$?

Ordering of Rules

- The ordering of rules are immaterial for the addition example, but are critical to the workings of limited subtraction.
- In fact, if we ignore the order and just allow any applicable rule to be used we get a form of non-determinism that makes these systems equivalent to Petri nets.
- The ordered kind are deterministic and are equivalent to a Petri net in which the transitions are prioritized.

Why Deterministic?

To see why determinism makes a difference, consider

$$3 \cdot 5x \rightarrow x$$

$$3x \rightarrow 2x$$

$$5x \rightarrow x$$

Starting with $135 = 3^3 5^1$, deterministically we get

$$135 \Rightarrow 9 \Rightarrow 6 \Rightarrow 4 = 2^2$$

Non-deterministically we get a larger, less selective set.

$$135 \Rightarrow 9 \Rightarrow 6 \Rightarrow 4 = 2^2$$

$$135 \Rightarrow 90 \Rightarrow 60 \Rightarrow 40 \Rightarrow 8 = 2^3$$

$$135 \Rightarrow 45 \Rightarrow 3 \Rightarrow 2 = 2^1$$

$$135 \Rightarrow 45 \Rightarrow 15 \Rightarrow 1 = 2^0$$

$$135 \Rightarrow 45 \Rightarrow 15 \Rightarrow 5 \Rightarrow 1 = 2^0$$

$$135 \Rightarrow 45 \Rightarrow 15 \Rightarrow 3 \Rightarrow 2 = 2^1$$

$$135 \Rightarrow 45 \Rightarrow 9 \Rightarrow 6 \Rightarrow 4 = 2^2$$

$$135 \Rightarrow 90 \Rightarrow 60 \Rightarrow 40 \Rightarrow 8 = 2^3$$

...

This computes 2^z where $0 \leq z \leq \text{desired answer}$. Think about it.

More on Determinism

In general, we might get an infinite set using non-determinism, whereas determinism might produce a finite set. To see this consider a system

$$2x \rightarrow x$$

$$2x \rightarrow 4x$$

starting with the number **2**.

Sample RM and FRS

Present a Register Machine that computes IsOdd. Assume $R2=x$; at termination, set $R1=1$ if x is odd; 0 otherwise.

1. DEC2[2, 4]
2. DEC2[1, 3]
3. INC1[4]
- 4.

Present a Factor Replacement System that computes IsOdd. Assume starting number is 3^x ; at termination, result is $2=2^1$ if x is odd; $1=2^0$ otherwise.

$3 \cdot 3^x \rightarrow x$

$3^x \rightarrow 2^x$

Sample FRS

Present a Factor Replacement System that computes IsPowerOf2. Assume starting number is $3^x 5$; at termination, result is $2=2^1$ if x is a power of 2; $1=2^0$ otherwise

$$3^{2*5} x \rightarrow 5*7 x$$

$$3*5*7 x \rightarrow x$$

$$3*5 x \rightarrow 2 x$$

$$5*7 x \rightarrow 7*11 x$$

$$7*11 x \rightarrow 3*11 x$$

$$11 x \rightarrow 5 x$$

$$5 x \rightarrow x$$

$$7 x \rightarrow x$$

Primitive Recursive

An Incomplete Model

Basis of PRFs

- The primitive recursive functions are defined by starting with some base set of functions and then expanding this set via rules that create new primitive recursive functions from old ones.

- The **base functions** are:

$C_a(x_1, \dots, x_n) = a$: constant functions

$I_i^n(x_1, \dots, x_n) = x_i$: identity functions

: aka projection

$S(x) = x+1$: an increment function

Building New Functions

- **Composition:**

If \mathbf{G} , \mathbf{H}_1 , \dots , \mathbf{H}_k are already known to be primitive recursive, then so is \mathbf{F} , where

$$\mathbf{F}(\mathbf{x}_1, \dots, \mathbf{x}_n) = \mathbf{G}(\mathbf{H}_1(\mathbf{x}_1, \dots, \mathbf{x}_n), \dots, \mathbf{H}_k(\mathbf{x}_1, \dots, \mathbf{x}_n))$$

- **Iteration (aka primitive recursion):**

If \mathbf{G} , \mathbf{H} are already known to be primitive recursive, then so is \mathbf{F} , where

$$\mathbf{F}(0, \mathbf{x}_1, \dots, \mathbf{x}_n) = \mathbf{G}(\mathbf{x}_1, \dots, \mathbf{x}_n)$$

$$\mathbf{F}(\mathbf{y}+1, \mathbf{x}_1, \dots, \mathbf{x}_n) = \mathbf{H}(\mathbf{y}, \mathbf{x}_1, \dots, \mathbf{x}_n, \mathbf{F}(\mathbf{y}, \mathbf{x}_1, \dots, \mathbf{x}_n))$$

We also allow definitions like the above, except iterating on \mathbf{y} as the last, rather than first argument.

Addition & Multiplication

Example: Addition

$$+(0,y) = \mathbf{I}_1^1(y)$$

$$+(x+1,y) = \mathbf{H}(x,y,+(x,y))$$

$$\text{where } \mathbf{H}(a,b,c) = \mathbf{S}(\mathbf{I}_3^3(a,b,c))$$

Example: Multiplication

$$*(0,y) = \mathbf{C}_0(y)$$

$$*(x+1,y) = \mathbf{H}(x,y,*(x,y))$$

$$\begin{aligned} \text{where } \mathbf{H}(a,b,c) &= +(\mathbf{I}_2^2(a,b,c), \mathbf{I}_3^3(a,b,c)) \\ &= \mathbf{b+c} = \mathbf{y} + *(x,y) = \mathbf{(x+1)*y} \end{aligned}$$

Intuitive Composition

- Any time you have already shown some functions to be primitive recursive, you can show others are by building them up through composition
- Example#1: If g and h are primitive recursive functions (prf) then so is $f(x) = g(h(x))$. As an explicit example $\text{Add2}(x) = S(S(x)) = x+2$ is a prf
- Example#2: This can also involve multiple functions and multiple arguments like, if g , h and j are prf's then so is $f(x,y) = g(h(x), j(y))$
The problem with giving an explicit example here is that interesting compositions tend to also involve induction.

Intuitive Inductions

- A function **F** can be defined inductively using existing prf's. Typically, we have one used for the basis and another for building inductively.
- Example#1: We can build addition from successor (S)
 $x+0 = x$ (formally $+(x,0) = I(x)$)
 $x+y+1 = S(x+y)$ (formally $+(x,y+1) = S(+(x,y))$)
- Example#2: We can build multiplication from addition
 $x*0 = 0$ (formally $*(x,0) = C_0$)
 $x*(y+1) = +(x,x*y)$ (formally $*(x,y+1) = +(x,*(x,y))$)

Basic Arithmetic

$x + 1$:

$$x + 1 = S(x)$$

$x - 1$:

$$0 - 1 = 0$$

$$(x+1) - 1 = x$$

$x + y$:

$$x + 0 = x$$

$$x + (y+1) = (x+y) + 1$$

$x - y$: // limited subtraction

$$x - 0 = x$$

$$x - (y+1) = (x-y) - 1$$

2nd Grade Arithmetic

$x * y$:

$$\mathbf{x * 0 = 0}$$

$$\mathbf{x * (y+1) = x*y + x}$$

$x!$:

$$\mathbf{0! = 1}$$

$$\mathbf{(x+1)! = (x+1) * x!}$$

Basic Relations

$x == 0:$

$$0 == 0 = 1$$

$$(y+1) == 0 = 0$$

$x == y:$

$$x == y = ((x - y) + (y - x)) == 0$$

$x \leq y :$

$$x \leq y = (x - y) == 0$$

$x \geq y:$

$$x \geq y = y \leq x$$

$x > y :$

$$x > y = \sim(x \leq y) \text{ /* See } \sim \text{ on next page */}$$

$x < y :$

$$x < y = \sim(x \geq y)$$

Basic Boolean Operations

$\sim x$:

$$\sim x = x == 0$$

signum(x): 1 if $x > 0$; 0 if $x == 0$

$$\sim(x == 0)$$

$x \ \&\& \ y$:

$$x \ \&\& \ y = \text{signum}(x * y)$$

$x \ || \ y$:

$$x \ || \ y = \sim((x == 0) \ \&\& \ (y == 0))$$

Definition by Cases

One case

$$f(x) = \begin{array}{ll} g(x) & \text{if } P(x) \\ h(x) & \text{otherwise} \end{array}$$

$$f(x) = P(x) * g(x) + (1-P(x)) * h(x)$$

Can use induction to prove this is true for all $k > 0$, where

$$f(x) = \begin{array}{ll} g_1(x) & \text{if } P_1(x) \\ g_2(x) & \text{if } P_2(x) \ \&\& \ \sim P_1(x) \\ \dots & \\ g_k(x) & \text{if } P_k(x) \ \&\& \ \sim(P_1(x) \ || \ \dots \ || \ \sim P_{k-1}(x)) \\ h(x) & \text{otherwise} \end{array}$$

Bounded Minimization 1

$f(x) = \mu z (z \leq x) [P(z)]$ if \exists such a z ,
 $= x+1$, otherwise

where $P(z)$ is primitive recursive.

Can show f is primitive recursive by

$$f(0) = 1 - P(0)$$

$$f(x+1) = f(x) \quad \text{if } f(x) \leq x$$
$$= x+2 - P(x+1) \quad \text{otherwise}$$

Bounded Minimization 2

$f(x) = \mu z (z < x) [P(z)]$ if \exists such a z ,
= x , otherwise

where $P(z)$ is primitive recursive.

Can show f is primitive recursive by

$$f(0) = 0$$

$$f(x+1) = \mu z (z \leq x) [P(z)]$$

Intermediate Arithmetic

$x // y$:

$x // 0 = 0$: silly, but want a value

$x // (y+1) = \mu z (z < x) [(z+1) * (y+1) > x]$

$x | y$: x is a divisor of y

$x | y = ((y // x) * x) == y$

Primality

firstFactor(x): first non-zero, non-one factor of **x**.

$$\text{firstfactor}(x) = \mu z (2 \leq z \leq x) [z|x], \\ 0 \text{ if none}$$

isPrime(x):

$$\text{isPrime}(x) = \text{firstFactor}(x) == x \ \&\& \ (x > 1)$$

prime(i) = i-th prime:

$$\text{prime}(0) = 2$$

$$\text{prime}(x+1) = \mu z (\text{prime}(x) < z \leq \text{prime}(x)! + 1) [\text{isPrime}(z)]$$

We will abbreviate this as **p_i** for **prime(i)**

Exponents

x^y :

$$x^0 = 1$$

$$x^{(y+1)} = x * x^y$$

$\text{exp}(x,i)$: the exponent of p_i in number x .

$$\text{exp}(x,i) = \mu z \ (z < x) \ [\sim(p_i^{(z+1)} \mid x)]$$

Pairing Functions

- $\text{pair}(x,y) = \langle x,y \rangle = 2^x (2y + 1) - 1$
- with inverses
$$\langle z \rangle_1 = \text{exp}(z+1,0)$$
$$\langle z \rangle_2 = (((z + 1) // 2^{\langle z \rangle_1}) - 1) // 2$$
- These are very useful and can be extended to encode **n**-tuples
$$\langle x,y,z \rangle = \langle x, \langle y,z \rangle \rangle \text{ (note: stack analogy)}$$

Pairing Function is 1-1 Onto

Prove that the pairing function $\langle x, y \rangle = 2^x (2y + 1) - 1$ is 1-1 onto the natural numbers.

Approach 1:

We will look at two cases, where we use the following modification of the pairing function, $\langle x, y \rangle + 1$, which implies the problem of mapping the pairing function to \mathbb{Z}^+ .

Case 1 (x=0)

Case 1:

For $x = 0$, $\langle 0, y \rangle + 1 = 2^0(2y+1) = 2y+1$. But every odd number is by definition one of the form $2y+1$, where $y \geq 0$; moreover, a particular value of y is uniquely associated with each such odd number and no odd number is produced by $2^x(2y+1)$ when $x > 0$. Thus, $\langle 0, y \rangle + 1$ is 1-1 onto the odd natural numbers.

Case 2 ($x > 0$)

Case 2:

For $x > 0$, $\langle x, y \rangle + 1 = 2^x(2y+1)$, where $2y+1$ ranges over all odd number and is uniquely associated with one based on the value of y (we saw that in case 1). 2^x must be even, since it has a factor of 2 and hence $2^x(2y+1)$ is also even. Moreover, from elementary number theory, we know that every even number except zero is of the form $2^x z$, where $x > 0$, z is an odd number and this pair x, z is unique. Thus, $\langle x, y \rangle + 1$ is 1-1 onto the even natural numbers, when $x > 0$.

The above shows that $\langle x, y \rangle + 1$ is 1-1 onto \mathbb{Z}^+ , but then $\langle x, y \rangle$ is 1-1 onto \mathbb{N} , as was desired.

μ Recursive

4th Model

A Simple Extension to Primitive
Recursive

μ Recursive Concepts

- All primitive recursive functions are algorithms since the only iterator is bounded. That's a clear limitation.
- There are algorithms like Ackerman's function that cannot be represented by the class of primitive recursive functions.
- The class of recursive functions adds one more iterator, the minimization operator (μ), read "the least value such that."

Ackermann's Function

- $A(1, j) = 2^j$ for $j \geq 1$
- $A(i, 1) = A(i-1, 2)$ for $i \geq 2$
- $A(i, j) = A(i-1, A(i, j-1))$ for $i, j \geq 2$
- Wilhelm Ackermann observed in 1928 that this is not a primitive recursive function.
- Ackermann's function grows too fast to have a for-loop implementation.
- The inverse of Ackermann's function is important to analyze Union/Find algorithm. Note: $A(4,4)$ is a super exponential number involving six levels of exponentiation. $\alpha(n) = A^{-1}(n, n)$ grows so slowly that it is less than 5 for any value of n that can be written using the number of atoms in our universe.

Union/Find

- Start with a collection **S** of unrelated elements – singleton equivalence classes
- **Union(x,y)**, **x** and **y** are in **S**, merges the class containing **x** (**[x]**) with that containing **y** (**[y]**)
- **Find(x)** returns the canonical element of **[x]**
- Can see if **x≡y**, by seeing if **Find(x)==Find(y)**
- How do we represent the classes?
- You should have learned that in CS2

The μ Operator

- Minimization:

If **G** is already known to be recursive, then so is **F**, where

$$\mathbf{F}(x_1, \dots, x_n) = \mu y (\mathbf{G}(y, x_1, \dots, x_n) == 1)$$

- We also allow other predicates besides testing for one. In fact any predicate that is recursive can be used as the stopping condition.

Universal Machine

- In the process of doing this reduction, we will build a Universal Machine.
- This is a single recursive function with two arguments. The first specifies the factor system (encoded) and the second the argument to this factor system.
- The Universal Machine will then simulate the given machine on the selected input.

Encoding FRS

- Let $(n, ((a_1, b_1), (a_2, b_2), \dots, (a_n, b_n)))$ be some factor replacement system, where (a_i, b_i) means that the i -th rule is

$$a_i x \rightarrow b_i x$$

- Encode this machine by the number F ,

$$2^n 3^{a_1} 5^{b_1} 7^{a_2} 11^{b_2} \dots p_{2n-1}^{a_n} p_{2n}^{b_n} p_{2n+1} p_{2n+2}$$

Simulation by Recursive # 1

- We can determine the rule of **F** that applies to **x** by

$$\mathbf{RULE(F, x) = \mu z (1 \leq z \leq \exp(F, 0)+1) [\exp(F, 2*z-1) | x]}$$

- Note: if **x** is divisible by **a_i**, and **i** is the least integer for which this is true, then **exp(F, 2*i-1) = a_i** where **a_i** is the number of prime factors of **F** involving **p_{2i-1}**. Thus, **RULE(F, x) = i**.

If **x** is not divisible by any **a_i**, **1 ≤ i ≤ n**, then **x** is divisible by **1**, and **RULE(F, x)** returns **n+1**. That's why we added **p_{2n+1} p_{2n+2}**.

- Given the function **RULE(F, x)**, we can determine **NEXT(F, x)**, the number that follows **x**, when using **F**, by

$$\mathbf{NEXT(F, x) = (x // \exp(F, 2*\mathbf{RULE(F, x)}-1)) * \exp(F, 2*\mathbf{RULE(F, x)})}$$

Simulation by Recursive # 2

- The configurations listed by F , when started on x , are

$$\text{CONFIG}(F, x, 0) = x$$

$$\text{CONFIG}(F, x, y+1) = \text{NEXT}(F, \text{CONFIG}(F, x, y))$$

- The number of the configuration on which F halts is

$$\text{HALT}(F, x) = \mu y [\text{CONFIG}(F, x, y) == \text{CONFIG}(F, x, y+1)]$$

This assumes we converge to a fixed point only if we stop

Simulation by Recursive # 3

- A Universal Machine that simulates an arbitrary Factor System, Turing Machine, Register Machine, Recursive Function can then be defined by
$$\text{Univ}(F, x) = \exp (\text{CONFIG} (F, x, \text{HALT} (F, x)), 0)$$
- This assumes that the answer will be returned as the exponent of the only even prime, **2**. We can fix **F** for any given Factor System that we wish to simulate.

Undecidability

We Can't Do It All

Computable Languages 1

Let's go over some important facts to this point:

1. Σ^* denotes the set of all strings over some finite alphabet Σ
2. $|\Sigma^*| = |\mathcal{N}|$, where \mathcal{N} is the set of natural numbers = the smallest infinite cardinal (the countable infinity)
3. A language L over Σ is a subset of Σ^* ; that is, $L \in \mathcal{P}(\Sigma^*) = 2^{\Sigma^*}$
Here \mathcal{P} denotes the power set constructor
4. $|L|$ is countable because $L \subseteq \Sigma^*$ (that is, $|L| \leq |\Sigma^*| = |\mathcal{N}|$)
5. $|\Sigma^*| < |\mathcal{P}(\Sigma^*)|$ (uncountable infinity) implies there are an uncountable number of languages over a given alphabet, Σ .
6. A program, P , in some programming language L , can be represented as a string over a finite alphabet, Σ_P that obeys the rules of constructing programs defined by L . As $P \in \Sigma_P^*$, there are at most a countably infinite number of programs that can be formed in the language L .

Computable Languages 2

7. Each program, P , in a programming language L , defines a function, $F_P: \Sigma_I^* \rightarrow \Sigma_O^*$ where Σ_I is the input alphabet and Σ_O is the output alphabet.
8. F_P defines an input language P_I for which F_P is defined (halts and produces an output). This is referred to as its domain in our terminology (Σ_I is its universe of discourse). The range of F_P , P_O , is the set of outputs. That is, $P_O = \{ y \mid \exists x \text{ in } P_I \text{ and } y = F_P(x) \}$
9. Since there are a countable number of programs, P , there can be at most a countable number of functions F_P and consequently, only a countable number of distinct input languages and output languages associated with programs in L_P . Thus, there are only a countable number of languages (input or output) that can be defined by any program, P .
10. But, there are an uncountable number of possible languages over any given alphabet – see 3 and 5.
11. Thus there must be languages over a given alphabet that have no descriptions – in terms of a program – or in terms of an algorithm. Thus, there are only a countably infinite number of languages that are computable among the uncountable number of possible languages.

Programming Languages

1. Programming languages that we use as software developers are in a sense “complete.” By complete we mean that they can be used to implement all procedures that we think are computable (definable by a computational model that we can “agree” covers all procedural activities).
2. **Challenge: Why did I say “agree” rather than “prove”?**
3. We mostly like programs that halt on all input (we call these algorithms), but we know it’s always possible to do otherwise in every programming language we think is complete (C, C++, C#, Java, Python, et al.)
4. We can, of course, define programming languages that define only algorithms.
5. Unfortunately, we cannot define a programming language that produces all and only algorithms (all and just programs that always halt).
6. The above (#5) is one of the main results shown in this course
7. However, before focusing on #5 we should recall that finite-state, push down and linear bounded automata are computational models that produce only algorithms (when we monitor the latter two for loops) – it’s just that these get us a subset of algorithms.

Classic Unsolvable Problem

Given an arbitrary program P , in some language L , and an input x to P , will P eventually stop when run with input x ?

The above problem is called the “Halting Problem.” Book denotes the Halting Problem as A_{TM} .

It is clearly an important and practical one – wouldn't it be nice to not be embarrassed by having your program run “forever” when you try to do a demo for the boss or professor? Unfortunately, there's a fly in the ointment as one can prove that no algorithm can be written in L that solves the halting problem for L .

Some terminology

We will say that a procedure, f , converges on input x if it eventually halts when it receives x as input. We denote this as $f(x)\downarrow$.

We will say that a procedure, f , diverges on input x if it never halts when it receives x as input. We denote this as $f(x)\uparrow$.

Of course, if $f(x)\downarrow$ then f defines a value for x . In fact we also say that $f(x)$ is defined if $f(x)\downarrow$ and undefined if $f(x)\uparrow$.

Finally, we define the domain of f as $\{x \mid f(x)\downarrow\}$.

The range of f is $\{y \mid \text{there exists an } x, f(x)\downarrow \text{ and } f(x) = y\}$.

Numbering Procedures

Any programming language needs to have an associated grammar that can be used to generate all legitimate programs.

By ordering the rules of the grammar in a way that generates programs in some lexical or syntactic order, we have a means to recursively enumerate the set of all programs. Thus, the set of procedures (programs) is re.

Using this fact, we will employ the notation that ϕ_x is the x -th procedure and $\phi_x(\mathbf{y})$ is the x -th procedure with input \mathbf{y} . We also refer to x as the procedure's index.

The universal machine

First, we can all agree that any complete model of computation must be able to simulate programs in its own language. We refer to such a simulator (interpreter) as the Universal machine, denote Univ. This program gets two inputs. The first is a description of the program to be simulated and the second of the input to that program. Since the set of programs in a model is re, we will assume both arguments are natural numbers; the first being the index of the program. Thus,

$$\text{Univ}(x,y) = \varphi_x(y)$$

Halting Problem (A_{TM})

Assume we can decide the halting problem. Then there exists some total function Halt such that

$$\text{Halt}(x,y) = \begin{cases} 1 & \text{if } \phi_x(\mathbf{y}) \text{ is defined} \\ 0 & \text{if } \phi_x(\mathbf{y}) \text{ is not defined} \end{cases}$$

Now we can view Halt as a mapping from N into N by treating its input as a single number representing the pairing of two numbers via the one-one onto function pair discussed earlier.

$$\text{pair}(x,y) = \langle x,y \rangle = 2^x (2y + 1) - 1$$

with inverses

$$\langle z \rangle_1 = \exp(z+1, 1)$$

$$\langle z \rangle_2 = (((z + 1) // 2^{\langle z \rangle_1}) - 1) // 2$$

The Contradiction

Now if Halt exist, then so does Disagree, where

$$\text{Disagree}(x) = \begin{cases} 0 & \text{if Halt}(x,x) = 0, \text{ i.e, if } \varphi_x(\mathbf{x}) \text{ is not defined} \\ \mu y (y == y+1) & \text{if Halt}(x,x) = 1, \text{ i.e, if } \varphi_x(\mathbf{x}) \text{ is defined} \end{cases}$$

Since Disagree is a program from \mathcal{N} into \mathcal{N} , Disagree can be reasoned about by Halt. Let d be such that $\text{Disagree} = [d]$, then

$$\begin{aligned} \text{Disagree}(d) \text{ is defined} & \Leftrightarrow \text{Halt}(d,d) = 0 \\ & \Leftrightarrow \varphi_d(\mathbf{d}) \text{ is undefined} \end{aligned}$$

$$\Leftrightarrow \text{Disagree}(d) \text{ is undefined}$$

But this means that Disagree contradicts its own existence. Since every step we took was constructive, except for the original assumption, we must presume that the original assumption was in error. Thus, the Halting Problem (A_{TM}) is not solvable.

Halting (A_{TM}) is recognizable

While the Halting Problem is not solvable, it is recognizable or semi-decidable.

To see this, consider the following semi-decision procedure. Let P be an arbitrary procedure and let x be an arbitrary natural number. Run the procedure P on input x until it stops. If it stops, say “yes.” If P does not stop, we will provide no answer. This semi-decides the Halting Problem. Here is a procedural description.

```
Semi_Decide_Halting() {  
    Read P, x;  
    P(x);  
    Print “yes”;  
}
```

Enumeration Theorem

- Define

$$\mathbf{W}_n = \{ \mathbf{x} \in \mathcal{N} \mid \varphi(\mathbf{n}, \mathbf{x}) \downarrow \}$$

- Theorem: A set \mathbf{B} is re iff there exists an \mathbf{n} such that $\mathbf{B} = \mathbf{W}_n$.

Proof: Follows from definition of $\varphi(\mathbf{n}, \mathbf{x})$.

- This gives us a way to enumerate the recursively enumerable (semi-decidable) sets.

Non-re Problems

- There are even “practical” problems that are worse than unsolvable -- they’re not even semi-decidable.
- The classic non-re problem is the Uniform Halting Problem, that is, the problem to decide of an arbitrary effective procedure P , whether or not P is an algorithm.
- Assume that the set of algorithms (TOTAL) can be enumerated, and that F accomplishes this. Then

$$F(x) = F_x$$

where F_0, F_1, F_2, \dots is a list of indexes of all and only the algorithms

The Contradiction

- Define $G(x) = \text{Univ}(F(x), x) + 1 = \varphi_{F(x)}(x) = F_x(x) + 1$

- But then G is itself an algorithm. Assume it is the g -th one

$$F(g) = F_g = G$$

Then, $G(g) = F_g(g) + 1 = G(g) + 1$

- But then G contradicts its own existence since G would need to be an algorithm.
- This cannot be used to show that the effective procedures are non-enumerable, since the above is not a contradiction when $G(g)$ is undefined. In fact, we already have shown how to enumerate the (partial) recursive functions.

The Set TOTAL

- The listing of all algorithms can be viewed as
as

$$\text{TOTAL} = \{ f \in \mathcal{N} \mid \forall x \varphi_f(x) \downarrow \}$$

- We can also note that

$$\text{TOTAL} = \{ f \in \mathcal{N} \mid W_f = \mathcal{N} \}, \text{ where } W_f \text{ is the domain of } \varphi_f$$

- Theorem: TOTAL is not re.
Proof: Shown earlier.

Consequences

- To capture all the algorithms, any model of computation must include some procedures that are not algorithms.
- Since the potential for non-termination is required, every complete model must have some form of iteration that is potentially unbounded.
- This means that simple, well-behaved for-loops (the kind where you can predict the number of iterations on entry to the loop) are not sufficient. While type loops are needed, even if implicit rather than explicit.

Insights

Non-re nature of algorithms

- No generative system (e.g., grammar) can produce descriptions of all and only algorithms
- No parsing system (even one that rejects by divergence) can accept all and only algorithms
- Of course, if you buy Church's Theorem, the set of all procedures can be generated. In fact, we can build an algorithmic acceptor of such programs.

Many unbounded ways

- How do you achieve divergence, i.e., what are the various means of unbounded computation in each of our models?
- GOTO: Turing Machines and Register Machines
- Minimization: Recursive Functions
 - Why not primitive recursion/iteration?
- Fixed Point: (Ordered) Factor Replacement Systems

Non-determinism

- It sometimes doesn't matter
 - Turing Machines, Finite-State Automata, Linear Bounded Automata
- It sometimes helps
 - Push Down Automata
- It sometimes hinders
 - Factor Replacement Systems, Petri Nets

Reducibility

Reduction Concepts

- Proofs by contradiction are tedious after you've seen a few. We really would like proofs that build on known unsolvable problems to show other, open problems are unsolvable. The technique commonly used is called reduction. It starts with some known unsolvable problem and then shows that this problem is no harder than some open problem in which we are interested.

Reduction Example#1

- We can show that the Halting Problem is no harder than the Uniform Halting Problem. Since we already know that the Halting Problem is unsolvable, we would now know that the Uniform Halting Problem is also unsolvable. We cannot reduce in the other direction since the Uniform Halting Problem is in fact harder.
- Let F be some arbitrary effective procedure and let x be some arbitrary natural number.
- Define $F_x(y) = F(x)$, for all $y \in \mathcal{N}$
- Then F_x is an algorithm if and only if F halts on x .
- Thus a solution to the Uniform Halting Problem (TOTAL) would provide a solution to the Halting Problem (HALT).

Reduction Examples #2 & #3

In all cases below we are assuming our variables are over \mathbb{N} .

HALT = $\{ \langle f, x \rangle \mid \varphi_f(x) \downarrow \}$ is unsolvable (undecidable, non-recursive)
TOTAL = $\{ f \mid \forall x \varphi_f(x) \downarrow \} = \{ f \mid W_f = \mathbb{N} \}$ is not even recursively enumerable (re, semidecidable)

- Show ZERO = $\{ f \mid \forall x \varphi_f(x) = 0 \}$ is unsolvable.
 $\langle f, x \rangle \in \text{HALT}$ iff $g(y) = \varphi_f(x) - \varphi_f(x)$ is zero for all y .
Thus, $\langle f, x \rangle \in \text{HALT}$ iff $g \in \text{ZERO}$ (really the index of g).
A solution to ZERO implies one for HALT, so ZERO is unsolvable.
- Show ZERO = $\{ f \mid \forall x \varphi_f(x) = 0 \}$ is non-re.
 $f \in \text{TOTAL}$ iff $h(x) = \varphi_f(x) - \varphi_f(x)$ is zero for all x .
Thus, $f \in \text{TOTAL}$ iff $h \in \text{ZERO}$ (really the index of h).
A semi-decision procedure for ZERO implies one for TOTAL, so ZERO is non-re.

Reduction and Equivalence

m-1, 1-1, Turing Degrees

Many-One Reduction

- Let A and B be two sets.
- We say A many-one reduces to B , $A \leq_m B$, if there exists an algorithm f such that $x \in A \Leftrightarrow f(x) \in B$
- We say that A is many-one equivalent to B , $A \equiv_m B$, if $A \leq_m B$ and $B \leq_m A$
- Sets that are many-one equivalent are in some sense equally hard or easy.

Many-One Degrees

- The relationship $A \equiv_m B$ is an equivalence relationship (why?)
- If $A \equiv_m B$, we say A and B are of the same many-one degree (of unsolvability).
- Decidable problems occupy three $m-1$ degrees: \emptyset , N , all others.
- The hierarchy of undecidable $m-1$ degrees is an infinite lattice (I'll discuss in class)

One-One Reduction

- Let A and B be two sets.
- We say A one-one reduces to B , $A \leq_1 B$, if there exists a 1-1 algorithm f such that $x \in A \Leftrightarrow f(x) \in B$
- We say that A is one-one equivalent to B , $A \equiv_1 B$, if $A \leq_1 B$ and $B \leq_1 A$
- Sets that are one-one equivalent are in a strong sense equally hard or easy.

One-One Degrees

- The relationship $A \equiv_1 B$ is an equivalence relationship (why?)
- If $A \equiv_1 B$, we say A and B are of the same one-one degree (of unsolvability).
- Decidable problems occupy infinitely many 1-1 degrees: each cardinality defines another 1-1 degree (think about it).
- The hierarchy of undecidable 1-1 degrees is an infinite lattice.

Turing (Oracle) Reduction

- Let A and B be two sets.
- We say A Turing reduces to B , $A \leq_t B$, if the existence of an oracle for B would provide us with a decision procedure for A .
- We say that A is Turing equivalent to B , $A \equiv_t B$, if $A \leq_t B$ and $B \leq_t A$
- Sets that are Turing equivalent are in a very loose sense equally hard or easy.

Turing Degrees

- The relationship $A \equiv_t B$ is an equivalence relationship (why?)
- If $A \equiv_t B$, we say A and B are of the same Turing degree (of unsolvability).
- Decidable problems occupy one Turing degree. We really don't even need the oracle.
- The hierarchy of undecidable Turing degrees is an infinite lattice.

Complete re Sets

- An re set C is re 1-1 (m-1, Turing) complete if, for any re set A , $A \leq_1 (\leq_m, \leq_t) C$.
- The set HALT is an re complete set (in regard to 1-1, m-1 and Turing reducibility).
- The re complete degree (in each sense of degree) sits at the top of the lattice of re degrees.

The Set Halt = K_0

- Halt = $K_0 = \{ \langle f, x \rangle \mid \varphi_f(x) \text{ is defined} \}$
- Let A be an arbitrary re set. By definition, there exists an effective procedure φ_a , such that $\text{dom}(\varphi_a) = A$. Put equivalently, there exists an index, a , such that $A = W_a$.
- $x \in A$ iff $x \in \text{dom}(\varphi_a)$ iff $\varphi_a(x) \downarrow$ iff $\langle a, x \rangle \in K_0$
- The above provides a 1-1 function that reduces A to K_0 ($A \leq_1 K_0$)
- Thus the universal set, Halt = K_0 , is an re (1-1, m-1, Turing) complete set.

The Set K

- $K = \{ f \mid \varphi_f(f) \text{ is defined} \}$
- Define $f_x(y)$ by $\forall y f_x(y) = \varphi_f(x)$. Let the index of f_x be f_x . (Yeah, that's overloading.)
- $\langle f, x \rangle \in K_0$ iff $x \in \text{dom}(\varphi_f)$ iff $\forall y[\varphi_{f_x}(y) \downarrow]$ implies $f_x \in K$.
- $\langle f, x \rangle \notin K_0$ iff $x \notin \text{dom}(\varphi_f)$ iff $\forall y[\varphi_{f_x}(y) \uparrow]$ implies $f_x \notin K$.
- The above provides a 1-1 function that reduces K_0 to K .
- Since K_0 is an re (1-1, m-1, Turing) complete set and K is re, then K is also re (1-1, m-1, Turing) complete.

Reduction and Rice's

Either Trivial or Undecidable

- Let P be some set of re languages, e.g. $P = \{ L \mid L \text{ is infinite re} \}$.
- We call P a property of re languages since it divides the class of all re languages into two subsets, those having property P and those not having property P .
- P is said to be trivial if it is empty (this is not the same as saying P contains the empty set) or contains all re languages.
- Trivial properties are not very discriminating in the way they divide up the re languages (all or nothing).

Rice's Theorem

Rice's Theorem: Let P be some non-trivial property of the re languages. Then

$L_P = \{ x \mid \text{dom } [x] \text{ is in } P \text{ (has property } P) \}$
is undecidable.

Note that membership in L_P is based purely on the domain of a function, not on any aspect of its implementation.

Rice's Proof-1

Proof: We will assume, *wlog*, that \mathbf{P} does not contain \emptyset . If it does we switch our attention to the complement of \mathbf{P} . Now, since \mathbf{P} is non-trivial, there exists some language \mathbf{L} with property \mathbf{P} . Let $[r]$ be a recursive function whose domain is \mathbf{L} (r is the index of a semi-decision procedure for \mathbf{L}). Suppose \mathbf{P} were decidable. We will use this decision procedure and the existence of r to decide \mathbf{K}_0 .

Rice's Proof-2

First we define a function $F_{r,x,y}$ for r and each function φ_x and input y as follows.

$$F_{r,x,y}(z) = \varphi(x, y) + \varphi(r, z)$$

The domain of this function is L if $\varphi_x(y)$ converges, otherwise it's \emptyset . Now if we can determine membership in L_P , we can use this algorithm to decide K_0 merely by applying it to $F_{r,x,y}$. An answer as to whether or not $F_{r,x,y}$ has property P is also the correct answer as to whether or not $\varphi_x(y)$ converges.

Rice's Proof-3

Thus, there can be no decision procedure for **P**. And consequently, there can be no decision procedure for any non-trivial property of re languages.

Note: This does not apply if **P** is trivial, nor does it apply if **P** can differentiate indices that converge for precisely the same values.

I/O Properties

- An I/O property, \mathcal{P} , of indices of recursive function is one that cannot differentiate indices of functions that produce precisely the same value for each input.
- This means that if two indices, \mathbf{f} and \mathbf{g} , are such that $\varphi_{\mathbf{f}}$ and $\varphi_{\mathbf{g}}$ converge on the same inputs and, when they converge, produce precisely the same result, then both \mathbf{f} and \mathbf{g} must have property \mathcal{P} , or neither one has this property.
- Note that any I/O property of recursive function indices also defines a property of re languages, since the domains of functions with the same I/O behavior are equal. However, not all properties of re languages are I/O properties.

Strong Rice's Theorem

Rice's Theorem: Let \mathcal{P} be some non-trivial I/O property of the indices of recursive functions. Then

$$\mathbf{S}_{\mathcal{P}} = \{ \mathbf{x} \mid \varphi_{\mathbf{x}} \text{ has property } \mathcal{P} \}$$

is undecidable.

Note that membership in $\mathbf{S}_{\mathcal{P}}$ is based purely on the input/output behavior of a function, not on any aspect of its implementation.

Strong Rice's Proof

- Given \mathbf{x} , \mathbf{y} , \mathbf{r} , where \mathbf{r} is in the set

$$\mathbf{S}_{\mathcal{P}} = \{\mathbf{f} \mid \varphi_{\mathbf{f}} \text{ has property } \mathcal{P}\},$$

define the function

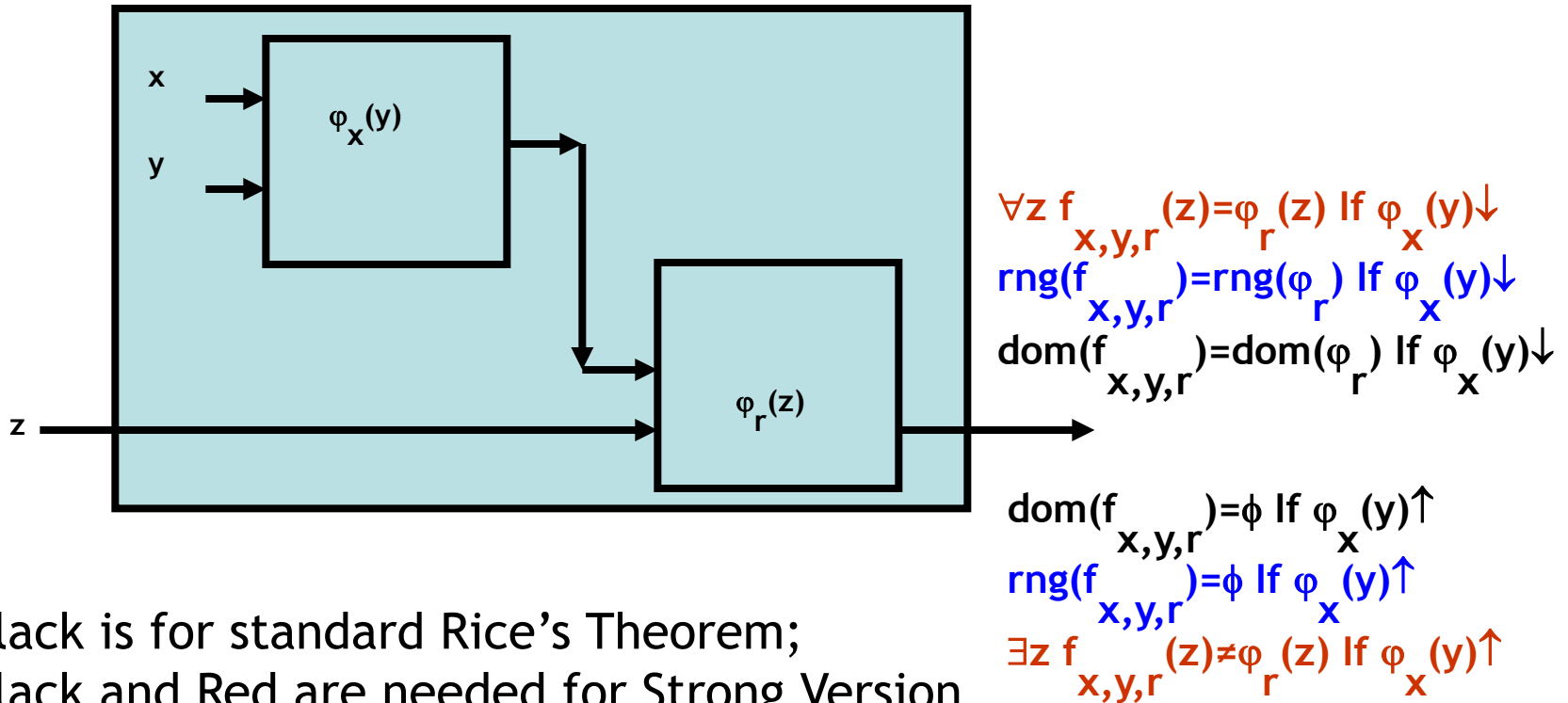
$$\mathbf{f}_{\mathbf{x},\mathbf{y},\mathbf{r}}(\mathbf{z}) = \varphi_{\mathbf{x}}(\mathbf{y}) - \varphi_{\mathbf{x}}(\mathbf{y}) + \varphi_{\mathbf{r}}(\mathbf{z}).$$

- $\mathbf{f}_{\mathbf{x},\mathbf{y},\mathbf{r}}(\mathbf{z}) = \varphi_{\mathbf{r}}(\mathbf{z})$ if $\varphi_{\mathbf{x}}(\mathbf{y}) \downarrow$; $= \phi$ if $\varphi_{\mathbf{x}}(\mathbf{y}) \uparrow$.

Thus, $\varphi_{\mathbf{x}}(\mathbf{y}) \downarrow$ iff $\mathbf{f}_{\mathbf{x},\mathbf{y},\mathbf{r}}$ has property \mathcal{P} , and so

$$\mathbf{K}_0 \leq \mathbf{S}_{\mathcal{P}}.$$

Rice's Picture Proof



Black is for standard Rice's Theorem;
 Black and Red are needed for Strong Version
 Blue is just another version based on range

Weak Rice's Theorems

Weak Rice's Theorem1: Let \mathcal{P} be some non-trivial I/O property of the indices of recursive functions. Then

$$\mathbf{S_{\mathcal{P}} = \{ x \mid \text{dom}(\varphi_x) \text{ has property } \mathcal{P} \}}$$

is undecidable.

$$\mathbf{\text{dom}(f_{x,y,r}) = \text{dom}(\varphi_r) \text{ if } \varphi_x(y) \downarrow ; = \phi \text{ if } \varphi_x(y) \uparrow}$$

Weak Rice's Theorem2: Let \mathcal{P} be some non-trivial I/O property of the indices of recursive functions. Then

$$\mathbf{S_{\mathcal{P}} = \{ x \mid \text{range}(\varphi_x) \text{ has property } \mathcal{P} \}}$$

is undecidable.

$$\mathbf{\text{range}(f_{x,y,r}) = \text{range}(\varphi_r) \text{ if } \varphi_x(y) \downarrow ; = \phi \text{ if } \varphi_x(y) \uparrow}$$

STP Predicate/VALUE Function

- **STP(f,x1,...,xn,t)** is a predicate defined to be true iff $\phi_f(x1, \dots, xn)$ converges in at most **t** steps.
- **STP** can be shown to be a prf, e.g.,
STP(f,x1,...,xn,t) = CONFIG(f,x1,...,xn,t) == CONFIG(f,x1,...,xn,t+1)
- **VALUE(f,x1,...,xn,t)** is a function that is meaningful only if **STP(f,x1,...,xn,t)**. In this case, it is **f(x1,...,xn)**. If \sim **STP(f, x1,...,xn, t)** then **VALUE(f, x1,...,xn, t)** is defined but meaningless. **VALUE** is also a prf, e.g.,
VALUE(f,x1,...,xn,t) = exp(CONFIG(f,x1,...,xn,t), 0)

Assignment # 8

Known Results:

HALT = $\{ \langle f, x \rangle \mid f(x) \downarrow \}$ is re (semi-decidable) but undecidable

TOTAL = $\{ f \mid \forall x f(x) \downarrow \}$ is non-re (not even semi-decidable)

1. Use reduction from **HALT** to show that one cannot decide **HasOdd** where **HasOdd** = $\{ f \mid \text{range}(f) \text{ contains an odd number} \}$
2. Show that **HasOdd** reduces to **HALT**. (1 plus 2 show they are equally hard)
3. Use Reduction from **TOTAL** to show that **IsAllOdds** is not even re, where **IsAllOdds** = $\{ f \mid \text{range}(f) = \text{Set of all odd natural numbers} \}$
4. Show **IsAllOdd** reduces to **TOTAL**. (3 plus 4 show they are equally hard)
5. Use Rice's Theorem to show that **HasOdd** is undecidable
6. Use Rice's Theorem to show that **IsAllOdd** is undecidable

Due: Thursday, Nov. 14, 11:59 PM (use Webcourses to turn in)

Recursively Enumerable

Properties of re Sets

Definition of re

- Some texts define re in the same way as I have defined semi-decidable.

$S \subseteq N$ is semi-decidable iff there exists a partially computable function g where

$$S = \{ x \in N \mid g(x) \downarrow \}$$

- I prefer the definition of re that says $S \subseteq N$ is re iff $S = \emptyset$ or there exists an algorithm f where

$$S = \{ y \mid \exists x f(x) == y \}$$

- We will prove these equivalent. Actually, f can be a primitive recursive function. (described briefly in class)

Semi-Decidable Implies re

Theorem: Let \mathbf{S} be semi-decided by \mathbf{G}_S . Assume \mathbf{G}_S is the g_S function in our enumeration of effective procedures. If $\mathbf{S} = \emptyset$ then \mathbf{S} is re by definition, so we will assume wlog that there is some $a \in \mathbf{S}$. Define the enumerating algorithm F_S by

$$F_S(\langle x, t \rangle) = \quad x * STP(g_S, x, t) \\ \quad \quad \quad + a * (1 - STP(g_S, x, t))$$

Note: F_S is primitive recursive and it enumerates every value in \mathbf{S} infinitely often.

re Implies Semi-Decidable

Theorem: By definition, S is re iff $S == \emptyset$ or there exists an algorithm F_S , over the natural numbers \mathbb{N} , whose range is exactly S . Define

$$\psi_S(x) = \begin{cases} \exists y [y == y+1], & \text{if } S == \emptyset \\ \exists y [F_S(y) == x], & \text{otherwise} \end{cases}$$

This achieves our result as the domain of ψ_S is the range of F_S , or empty if $S == \emptyset$.

Domain of a Procedure

Corollary: \mathbf{S} is re/semi-decidable iff \mathbf{S} is the domain / range of a partial recursive predicate \mathbf{F}_S .

Proof: The predicate ψ_S we defined earlier to semi-decide \mathbf{S} , given its enumerating function, can be easily adapted to have this property.

$$\psi_S(x) = \begin{cases} \exists y [y == y+1], & \text{if } \mathbf{S} == \emptyset \\ x^*(\exists y [F_S(y) == x]), & \text{otherwise} \end{cases}$$

Recursive Implies re

Theorem: Recursive implies re.

Proof: **S** is recursive implies there is an algorithm (predicate) χ_S (called the characteristic function for S) such that

$$\mathbf{S} = \{ \mathbf{x} \in \mathbf{N} \mid \chi_S(\mathbf{x}) \}$$

Define $\mathbf{g}_S(\mathbf{x}) = \exists \mathbf{y} (\chi_S(\mathbf{x}))$ – **diverges if false**

Clearly

$$\begin{aligned} \mathbf{dom}(\mathbf{g}_S) &= \{ \mathbf{x} \in \mathbf{N} \mid \mathbf{g}_S(\mathbf{x}) \downarrow \} \\ &= \{ \mathbf{x} \in \mathbf{N} \mid \chi_S(\mathbf{x}) \} \\ &= \mathbf{S} \end{aligned}$$

Related Results

Theorem: \mathbf{S} is re iff \mathbf{S} is semi-decidable.

Proof: That's what we proved.

Theorem: \mathbf{S} and $\sim\mathbf{S}$ are both re (semi-decidable)
iff \mathbf{S} (equivalently $\sim\mathbf{S}$) is recursive (decidable).

Proof: Let f_S semi-decide \mathbf{S} and $f_{\sim S}$ semi-decide $\sim\mathbf{S}$. We can decide \mathbf{S} by g_S

$$g_S(x) = \text{STP}(f_S, x, \mu t (\text{STP}(f_S, x, t) \parallel \text{STP}(f_{\sim S}, x, t)))$$

$\sim\mathbf{S}$ is decided by $g_{\sim S}(x) = \sim g_S(x) = 1 - g_S(x)$.

The other direction is immediate since, if \mathbf{S} is decidable then $\sim\mathbf{S}$ is decidable (just complement g_S) and hence they are both re (semi-decidable).

re Characterizations

Theorem: Suppose $S \neq \emptyset$ then the following are equivalent:

1. S is re
2. S is the range of a primitive rec. function
3. S is the range of a total recursive function
4. S is the domain of a partial rec. function
5. S is the range/domain of a partial rec. function whose domain is the same as its range and which acts as an identity when it converges

Quantification#1

- **S** is decidable iff there exists an algorithm χ_S (called **S**'s characteristic function) such that

$$\mathbf{x} \in \mathbf{S} \Leftrightarrow \chi_S(\mathbf{x})$$

This is just the definition of decidable.

- **S** is re iff there exists an algorithm \mathbf{A}_S where

$$\mathbf{x} \in \mathbf{S} \Leftrightarrow \exists t \mathbf{A}_S(\mathbf{x}, t)$$

This is clear since, if g_S is the index of a procedure that semi-decides **S**, then

$$\mathbf{x} \in \mathbf{S} \Leftrightarrow \exists t \mathbf{STP}(g_S, \mathbf{x}, t)$$

So, $\mathbf{A}_S(\mathbf{x}, t) = \mathbf{STP}_{g_S}(\mathbf{x}, t)$, where \mathbf{STP}_{g_S} is the **STP** function with its first argument fixed.

Quantification#2

- **S** is re iff there exists an algorithm A_S such that
$$\mathbf{x} \notin \mathbf{S} \Leftrightarrow \forall \mathbf{t} A_S(\mathbf{x}, \mathbf{t})$$
This is clear since, if g_S is the index of the procedure ψ_S that semi-decides **S**, then
$$\mathbf{x} \notin \mathbf{S} \Leftrightarrow \sim \exists \mathbf{t} \mathbf{STP}(g_S, \mathbf{x}, \mathbf{t}) \Leftrightarrow \forall \mathbf{t} \sim \mathbf{STP}(g_S, \mathbf{x}, \mathbf{t})$$
So, $A_S(\mathbf{x}, \mathbf{t}) = \sim \mathbf{STP}_{g_S}(\mathbf{x}, \mathbf{t})$, where \mathbf{STP}_{g_S} is the **STP** function with its first argument fixed.
- Note that this works even if **S** is recursive (decidable). The important thing there is that if **S** is recursive then it may be viewed in two normal forms, one with existential quantification and the other with universal quantification.
- The complement of an re set is **co-re**. A set is recursive (decidable) iff it is both re and co-re.

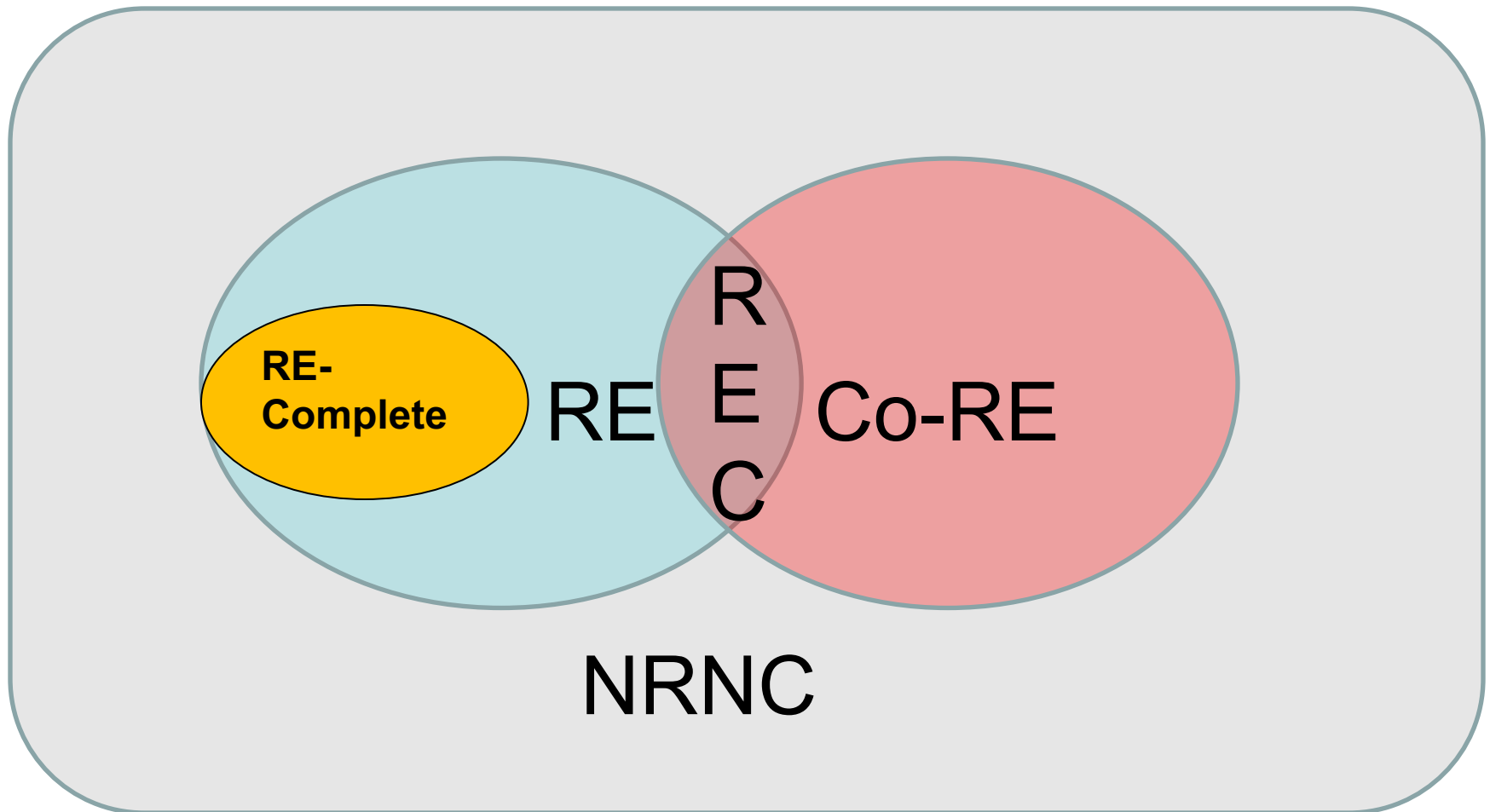
Quantification#3

- The **Uniform Halting Problem** was already shown to be non-re. It turns out its complement is also not re. In fact, we can (but won't) show that **TOTAL** requires an alternation of quantifiers. Specifically,

$$\mathbf{f} \in \mathbf{TOTAL} \Leftrightarrow \forall \mathbf{x} \exists \mathbf{t} (\mathbf{STP}(\mathbf{f}, \mathbf{x}, \mathbf{t}))$$

and this is the minimum quantification we can use, given that the quantified predicate is recursive.

UNIVERSE OF SETS



$$\text{NonRE} = (\text{NRNC} \cup \text{Co-RE}) - \text{REC}$$

Sample Question#1

1. Given that the predicate **STP** and the function **VALUE** are algorithms, show that we can semi-decide

HZ = { f | φ_f evaluates to 0 for some input }

Note: **STP(f, x, s)** is true iff $\varphi_f(\mathbf{x})$ converges in **s** or fewer steps and, if so, **VALUE(f, x, s) = $\varphi_f(\mathbf{x})$.**

Sample Questions#2,3

2. Use Rice's Theorem to show that **HZ** is undecidable, where **HZ** is

$$\mathbf{HZ} = \{ f \mid \varphi_f \text{ evaluates to } 0 \text{ for some input} \}$$

3. Redo using Reduction from **HALT**.

Sample Question#4

4. Let $\mathbf{P} = \{ f \mid \exists x [\mathbf{STP}(f, x, x)] \}$. Why does Rice's theorem not tell us anything about the undecidability of \mathbf{P} ?

Sample Question#5

5. Let **S** be an re (recursively enumerable), non-recursive set, and **T** be an re, possibly recursive non-empty set. Let

$$\mathbf{E} = \{ \mathbf{z} \mid \mathbf{z} = \mathbf{x} + \mathbf{y}, \text{ where } \mathbf{x} \in \mathbf{S} \text{ and } \mathbf{y} \in \mathbf{T} \}.$$

Answer with proofs, algorithms or counterexamples, as appropriate, each of the following questions:

- (a) Can **E** be non re?
- (b) Can **E** be re non-recursive?
- (c) Can **E** be recursive?

Assignment # 9

1. Use quantification of an algorithmic predicate to estimate the complexity (decidable, re, co-re, non-re) of each of the following, (a)-(d):
 - a) **HasDip** = { f | for some x, y , where $y > x$, $f(x) \downarrow$ and $f(y) \downarrow$ and $f(y) < f(x)$ }
 - b) **HasMax** = { f | for some x , where $f(x) \downarrow$, $f(y) < f(x)$, whenever $f(y) \downarrow$ }
 - c) **NotLarge** = { f | if $x \in \text{Range}(f)$ then $x < 100$ }
 - d) **ZeroStart** = { f | if $x < 100$ and $f(x) \downarrow$ in fewer than 100 steps, then $f(x) = 0$ }
2. Let set **A** be a non-empty recursive subset of \mathbb{N} , and let **B** be an re non-recursive subset of \mathbb{N} . Consider **C** = { z | $z = \max(x, y)$ where $x \in A$ & $y \in B$ }. For (a)-(c), either show sets **A** and **B** with the specified property or demonstrate that this property cannot hold.
 - a) **Can C be recursive?**
 - b) **Can C be re non-recursive (undecidable)?**
 - c) **Can C be non-re?**

Due: Thursday, Nov. 21, 11:59PM (use Webcourses to turn in)

Rewriting Systems

Semi-Thue Systems

- Devised by Emil Post based on earlier work by Axel Thue
- $S = (\Sigma, R)$, where Σ is a finite alphabet and R is a finite set of rules of form $\alpha_i \rightarrow \beta_i$, $\alpha_i, \beta_i \in \Sigma^*$
- We define \Rightarrow^* as the reflexive, transitive closure of \Rightarrow , where $w \Rightarrow x$ iff $w = y\alpha z$ and $x = y\beta z$, where $\alpha \rightarrow \beta$

Simulating Turing Machines

- Basically, we need at least one rule for each 4-tuple in the Turing machine's description.
- The rules lead from one instantaneous description to another.
- The Turing ID $\alpha qa\beta$ is represented by the string $\#\alpha qa\beta\#$, a being the scanned symbol.
- The tuple $q a b s$ leads to $qa \rightarrow sb$
- Moving right and left can be harder due to blanks and the requirement that α and β are minimum length strings containing all non-blanks.

Details of $\text{Halt(TM)} \leq \text{Word(ST)}$

- Let $M = (Q, \{0,1\}, T)$, T is Turing table.
- If $qabs \in T$, add rule $qa \rightarrow sb$
- If $qaRs \in T$, add rules
 - $q1b \rightarrow 1sb$ $a=1, \forall b \in \{0,1\}$
 - $q1\# \rightarrow 1s0\#$ $a=1$
 - $cq0b \rightarrow c0sb$ $a=0, \forall b,c \in \{0,1\}$
 - $\#q0b \rightarrow \#sb$ $a=0, \forall b \in \{0,1\}$
 - $cq0\# \rightarrow c0s0\#$ $a=0, \forall c \in \{0,1\}$
 - $\#q0\# \rightarrow \#s0\#$ $a=0$
- If $qaLs \in T$, add rules
 - $bqac \rightarrow sbac$ $\forall a,b,c \in \{0,1\}$
 - $\#qac \rightarrow \#s0ac$ $\forall a,c \in \{0,1\}$
 - $bq1\# \rightarrow sb1\#$ $a=1, \forall b \in \{0,1\}$
 - $\#q1\# \rightarrow \#s01\#$ $a=1$
 - $bq0\# \rightarrow sb\#$ $a=0, \forall b \in \{0,1\}$
 - $\#q0h \rightarrow \#s0\#$ $a=0$

Clean-Up

- Assume q_{init} is start state and only one accepting state exists q_{acc}
- We will start in $\#1^x q_{init} 0\#$, seeking to accept x (enter q_{acc}) or reject (run forever).
- Add rules
 - $q_{acc}a \rightarrow q_{acc} \quad \forall a \in \{0,1\}$
 - $bq_{acc} \rightarrow q_{acc} \quad \forall b \in \{0,1\}$
- The added rule allows us to “erase” the tape if we accept x .
- This means that acceptance can be changed to generating $\#q_{acc}\#$.
- The next slide shows the consequences.

Semi-Thue Word Problem

- Construction from TM, M , gets:
- $\#1^xq_{\text{init}}0\# \Rightarrow_{\Sigma(M)^*} \#q_{\text{acc}}\#$ iff $x \in \mathcal{L}(M)$.
- $\#q_{\text{acc}}\# \Rightarrow_{\Pi(M)^*} \#1^xq_{\text{init}}0\#$ iff $x \in \mathcal{L}(M)$.
- $\#q_{\text{acc}}\# \Leftrightarrow_{\Sigma(M)^*} \#1^xq_{\text{init}}0\#$ iff $x \in \mathcal{L}(M)$.
 - This is called a Thue system where rules can be applied in either direction ($\alpha \leftrightarrow \beta$)
- Can recast both Semi-Thue and Thue Systems to ones over alphabet $\{0,1\}$. That is, a binary alphabet is sufficient for undecidability.

More on Grammars

Grammars and re Sets

- Every grammar lists an re set.
- Some grammars (regular, CFL and CSG) produce recursive sets.
- Type 0 grammars are as powerful at generating (producing) re sets as Turing machines are at enumerating them (Proof later).

Post Correspondence Problem

- Many problems related to grammars can be shown to be no more complex than the Post Correspondence Problem (PCP).
- Each instance of PCP is denoted: Given $n > 0$, Σ a finite alphabet, and two n -tuples of words $(x_1, \dots, x_n), (y_1, \dots, y_n)$ over Σ , does there exist a sequence i_1, \dots, i_k , $k > 0$, $1 \leq i_j \leq n$, such that
$$x_{i_1} \dots x_{i_k} = y_{i_1} \dots y_{i_k} \quad ?$$
- Example of PCP:
 $n = 3, \Sigma = \{a, b\}, (aba, bb, a), (bab, b, baa)$.
Solution 2, 3, 1, 2
 $bbabaabb = bbaababb$
- In general, PCP is undecidable (no proof will be given)

ST(Word) \leq PCP

- Start with Semi-Thue System
 - $aba \rightarrow ab$; $a \rightarrow aa$; $b \rightarrow a$
 - Instance of word problem: $bbbb \Rightarrow^*? aa$
- Convert to PCP over $\{ [,], *, a, b, \underline{a}, \underline{b} \}$

$$\begin{array}{l}
 x = [bbbb^* \quad ab \quad \underline{ab} \quad aa \quad \underline{aa} \quad a \quad \underline{a} \quad] \\
 y = [\quad \underline{aba} \quad aba \quad \underline{a} \quad a \quad \underline{b} \quad b \quad \underline{*aa}] \\
 x' = \quad * \quad \underline{*} \quad a \quad \underline{a} \quad b \quad \underline{b} \\
 \underline{y'} = \underline{*} \quad \underline{*} \quad \underline{a} \quad a \quad \underline{b} \quad b
 \end{array}$$

How PCP Construction Works?

- Using underscored letters avoids solutions that don't relate to word problem instance.
E.g.,
aba a
ab aa
- Top row insures start with $[W_0^*$
- Bottom row insures end with $_*W_f]$
- Bottom row matches W_i , while top matches W_{i+1} (one is underscored)

PCP is undecidable

- The essential idea is that we can embed computational traces in instances of PCP, such that a solution exists if and only if the computation terminates.
- Such a construction shows that the Halting Problem is reducible to PCP and so PCP must also be undecidable.
- As we will see PCP can often be reduced to problems about grammars, showing those problems to also be undecidable.

Ambiguity of CFG

- Arbitrary instance of PCP,
 $\mathbf{P} = (\Sigma, n, ((x_1, \dots, x_n), (y_1, \dots, y_n)))$
- $\mathbf{G} = (\{S, A, B\}, \Sigma, R, S)$, where R is:
 - $S \rightarrow A \mid B$
 - $A \rightarrow x_i A [i] \mid x_i [i] \quad 1 \leq i \leq n$
 - $B \rightarrow y_i B [i] \mid y_i [i] \quad 1 \leq i \leq n$
 - $A \Rightarrow^* x_{i_1} \dots x_{i_k} [i_k] \dots [i_1] \quad k > 0$
 - $B \Rightarrow^* y_{i_1} \dots y_{i_k} [i_k] \dots [i_1] \quad k > 0$
- Ambiguous if and only if there is a solution to this PCP instance, \mathbf{P} .

Intersection of CFLs

- Problem to determine if arbitrary CFG's define overlapping languages
- Just take the grammar consisting of all the A-rules from previous, and a second grammar consisting of all the B-rules. Call the languages generated by these grammars, L_A and L_B .
 $L_A \cap L_B \neq \emptyset$, if and only there is a solution to this PCP instance.

Non-emptiness of CSL

- Arbitrary instance of PCP,
 $P = (\Sigma, n, ((x_1, \dots, x_n), (y_1, \dots, y_n)))$
- $G = (\{S, T\} \cup \Sigma, \{*\}, R, S)$, where R is:
 $S \rightarrow x_i S y_i^R \mid x_i T y_i^R \quad 1 \leq i \leq n$
 $a T a \rightarrow * T *$
 $* a \rightarrow a *$
 $a * \rightarrow * a$
 $T \rightarrow *$

CSG Produces Something

- Our only terminal in previous grammar is $*$. We get strings of form $*^{2j+1}$, for some j 's if and only if there is a solution to this PCP instance. Get \emptyset otherwise.
- Thus, \mathbf{P} has a solution iff
 - $\mathbf{L(G)} \neq \emptyset$
 - $\mathbf{L(G)}$ is infinite

Traces and Grammars

Traces (Valid Computations)

- A trace of a machine M , is a word of the form

X_0 # X_1 # X_2 # X_3 # ... # X_{k-1} # X_k

where $X_i \Rightarrow X_{i+1}$, $0 \leq i < k$, X_0 is a starting configuration and X_k is a terminating configuration.

- We allow some laxness, where the configurations might be encoded in a convenient manner. Many texts show that a context free grammar can be devised which approximates traces by either getting the even-odd pairs right, or the odd-even pairs right. The goal is then to intersect the two languages, so the result is a trace. This then allows us to create CFLs $L1$ and $L2$, where $L1 \cap L2 \neq \emptyset$, just in case the machine has an element in its domain. Since this is undecidable, the non-emptiness of the intersection problem is also undecidable. This is an alternate proof to one we already showed based on PCP.

Quotients of CFLs (concept)

Let $L1 = L(G1) = \{ \$ \# Y_0 \# Y_1 \# Y_2 \# Y_3 \# \dots \# Y_{2j} \# Y_{2j+1} \# \}$

where $Y_{2i} \Rightarrow Y_{2i+1}$, $0 \leq i \leq j$.

This checks the even/odd steps of an even length computation.

Now, let $L2=L(G2)=\{X_0 \$ \# X_0 \# X_1 \# X_2 \# X_3 \# X_4 \# \dots \# X_{2k-1} \# X_{2k} \# Z_0 \# \}$

where $X_{2i-1} \Rightarrow X_{2i}$, $1 \leq i \leq k$ and Z_0 is a unique halting configuration.

This checks the odd/steps of an even length computation and includes an extra copy of the starting number prior to its \$. Its final configuration is an accepting one

Now, consider the quotient of $L2 / L1$. The only ways a member of $L1$ can match a final substring in $L2$ is to line up the \$ signs. But then they serve to check out the validity and termination of the computation. Moreover, the quotient leaves only the starting point (the one on which the machine halts.) Thus,

$L2 / L1 = \{ X_0 \mid \text{the system halts} \}$.

Since deciding the members of an re set is in general undecidable, we have shown that membership in the quotient of two CFLs is also undecidable.

Finish Quotient

Now, consider the quotient of $L2 / L1$. The only ways a member of $L1$ can match a final substring in $L2$ is to line up the \$ signs. But then they serve to check out the validity and termination of the computation. Moreover, the quotient leaves only the starting number (the one on which the machine halts.) Thus,

$L2 / L1 = \{ X \mid \text{the system } F \text{ halts on zero} \}$.

Since deciding the members of an arbitrary set is in general undecidable, we have shown that membership in the quotient of two CFLs is also undecidable.

Traces and Type 0 (PSG)

- Here, it is actually easier to show a simulation of a Turing machine than of a Factor System.
- Assume we are given some machine M , with Turing table T (using Post notation). We assume a tape alphabet of Σ that includes a blank symbol B .
- Consider a starting configuration C_0 . Our rules will be

S	→	# C0 #	where C0 = Yq₀aX is initial ID
q a	→	s b	if q a b s ∈ T
b q a x	→	b a s x	if q a R s ∈ T, a,b,x ∈ Σ
b q a #	→	b a s B #	if q a R s ∈ T, a,b ∈ Σ
# q a x	→	# a s x	if q a R s ∈ T, a,x ∈ Σ, a≠B
# q a #	→	# a s B #	if q a R s ∈ T, a ∈ Σ, a≠B
# q a x	→	# s x #	if q a R s ∈ T, x ∈ Σ, a=B
# q a #	→	# s B #	if q a R s ∈ T, a=B
b q a x	→	s b a x	if q a L s ∈ T, a,b,x ∈ Σ
# q a x	→	# s B a x	if q a L s ∈ T, a,x ∈ Σ
b q a #	→	s b a #	if q a L s ∈ T, a,b ∈ Σ, a≠B
# q a #	→	# s B a #	if q a L s ∈ T, a ∈ Σ, a≠B
b q a #	→	s b #	if q a L s ∈ T, b ∈ Σ, a=B
# q a #	→	# s B #	if q a L s ∈ T, a=B
f	→	λ	if f is a final state
#	→	λ	just cleaning up the dirty linen

CSG and Undecidability

- We can almost do anything with a CSG that can be done with a Type 0 grammar. The only thing lacking is the ability to reduce lengths, but we can throw in a character that we think of as meaning “deleted”. Let’s use the letter d as a deleted character and use the letter e to mark both ends of a word.
- Let $G = (V, T, P, S)$ be an arbitrary Type 0 grammar.
- Define the CSG $G' = (V \cup \{S', D\}, T \cup \{d, e\}, S', P')$, where P' is

S'	\rightarrow	$e S e$	
$D x$	\rightarrow	$x D$	when $x \in V \cup T$
$D e$	\rightarrow	$e d$	push the delete characters to far right
α	\rightarrow	β	where $\alpha \rightarrow \beta \in P$ and $\alpha \leq \beta$
α	\rightarrow	βD^k	where $\alpha \rightarrow \beta \in P$ and $\alpha - \beta = k > 0$
- Clearly, $L(G') = \{ e w e d^m \mid w \in L(G) \text{ and } m \geq 0 \text{ is some integer} \}$
- For each $w \in L(G)$, we cannot, in general, determine for which values of m , $e w e d^m \in L(G')$. We would need to ask a potentially infinite number of questions of the form “does $e w e d^m \in L(G')$ ” to determine if $w \in L(G)$. That’s a semi-decision procedure.

Some Consequences

- CSGs are not closed under Init, Final, Mid, quotient with regular sets and homomorphism (okay for λ -free homomorphism)
- We also have that the emptiness problem is undecidable from this result. That gives us two proofs of this one result.
- For Type 0, emptiness and even the membership problems are undecidable.

Undecidability

- Is $L = \emptyset$, for CSL, L? **PCP reduction**
- Is $L = \Sigma^*$, for CFL (CSL), L? **Trace Complement**
- Is $L_1 = L_2$ for CFLs (CSLs), L_1, L_2 ? **$L_1 = \Sigma^*$**
- Is $L_1 \subseteq L_2$ for CFLs (CSLs), L_1, L_2 ? **$L_1 = \Sigma^*$**
- Is $L_1 \cap L_2 = \emptyset$ for CFLs (CSLs), L_1, L_2 ? **PCP reduction**
- Is L regular, for CFL (CSL), L? **Think about it**
- Is $L_1 \cap L_2$ a CFL for CFLs, L_1, L_2 ? **Think about it**
- Is $\sim L$ CFL, for CFL, L? **Think about it**

More Undecidability

- Is CFL, L , ambiguous? **PCP**
- Is $L=L^2$, L a CFL? **Will Do**
- Is L_1/L_2 finite, L_1 and L_2 CFLs?
Language is any RE set
- Membership in L_1/L_2 , L_1 and L_2 CFLs?
Language is any RE set

ST(Word) \leq PSL(Membership)

- Recast semi-Thue system making all symbols non-terminal, adding S and T to non-terminals and terminal set $\Sigma = \{a\}$

$$G: S \rightarrow \#1^{x_{q_{\text{init}}}}0\#$$

$$\#q_{\text{acc}}\# \rightarrow T$$

$$T \rightarrow aT$$

$$T \rightarrow \lambda$$

- $x \in \mathcal{L}(M)$ iff $\mathcal{L}(G) \neq \emptyset$ iff $\mathcal{L}(G)$ infinite
iff $\lambda \in \mathcal{L}(G)$ iff $a \in \mathcal{L}(G)$ iff $\mathcal{L}(G) = \Sigma^*$

Consequences for PSG

- Unsolvables
 - $\mathcal{L}(G) = \emptyset$
 - $\mathcal{L}(G) = \Sigma^*$
 - $\mathcal{L}(G)$ infinite
 - $w \in \mathcal{L}(G)$, for arbitrary w
 - $\mathcal{L}(G) \supseteq \mathcal{L}(G2)$
 - $\mathcal{L}(G) = \mathcal{L}(G2)$
- Latter two results follow when have
 - $G2: S \rightarrow aS \mid \lambda \quad a \in \Sigma \quad \mathcal{L}(G2) = \Sigma^*$

$$L = \Sigma^*?$$

- If L is regular, then $L = \Sigma^*$? is decidable
 - Easy – Reduce to minimal state deterministic FSA, \mathcal{A}_L accepting L . $L = \Sigma^*$ iff \mathcal{A}_L is a one-state machine, whose only state is accepting
- If L is context free, then $L = \Sigma^*$? is undecidable
 - The key here is that the complement of a Turing Machine's valid terminating traces is a CFL – requires just one error which is context free; requiring all pairs to be correct is a CSL

$$L(G) = L(G)^2?$$

- The problem to determine if $L = \Sigma^*$ is Turing reducible to the problem to decide if $L \bullet L \subseteq L$, so long as L is selected from a class of languages C over the alphabet Σ for which we can decide if $\Sigma \cup \{\lambda\} \subseteq L$.
- Corollary 1:
The problem “is $L \bullet L = L$, for L context free or context sensitive?” is undecidable

$L(G) = L(G)^2?$ is undecidable

- **Question: Does $L \bullet L$ get us anything new?**
 - i.e., Is $L \bullet L = L$?
- **Membership in a CFL is decidable.**
- **Claim is that $L = \Sigma^*$ iff**
 - (1) $\Sigma \cup \{\lambda\} \subseteq L$; and
 - (2) $L \bullet L = L$
- **Clearly, if $L = \Sigma^*$ then (1) and (2) trivially hold.**
- **Conversely, we have $\Sigma^* \subseteq L^* = \bigcup_{n \geq 0} L^n \subseteq L$**
 - first inclusion follows from (1); second from (2) as $L \bullet L = L$ implies $L^{n+1} = L^n$, $n > 0$

Computational Complexity

Limited to Concepts of P and NP

COT6410 covers much more

Complexity vs ..

- Complexity seeks to categorize problems as easy (polynomial) or hard (exponential or even worse). Some parts focus on time; others on space.
- Computability seeks to categorize problems as algorithmically solvable or not.
- Algorithm Design & Analysis tries to find the most efficient algorithms to solve specific problems.

Research Territory

Decidable – vs – Undecidable
(area of Computability Theory)

Exponential – vs – polynomial
(area of Computational Complexity)

Algorithms for any of these
(area of Algorithm Design/Analysis)

Decision vs Optimization

Two types of problems are of particular interest:

Decision Problems ("Yes/No" answers)

Optimization problems ("best" answers)

(there are other types)

Natural Pairs of Problems

Interestingly, these usually come in pairs

a decision problem, and

an optimization problem.

Equally easy, or equally difficult, to solve.

Both can be solved in polynomial time, or both require at least exponential time.

Very Hard Problems

- Some problems have no algorithm (e. g., Halting Problem.)
- No mechanical/logical procedure will ever solve all instances of any such problem!!
- Some problems have only exponential algorithms (provably so – they must take at least order 2^n steps) So far, only a few have been proven, but there may be many. We suspect so.
- Provably exponential include
 - Towers of Hanoi: we can prove that any algorithm that solves this problem must have a worst-case running time that is at least $2^n - 1$.
 - List all permutations (all possible orderings) of n numbers.

Easy Problems

Many problems have polynomial algorithms (Fortunately).

Why fortunately? Because, most exponential algorithms are essentially useless for problem instances with n much larger than 50 or 60. We have algorithms for them, but the best of these will take 100's of years to run, even on much faster computers than we now envision.

Three Classes of Problems

Problems proven to be in these three groups (classes) are, respectively,

Undecidable, Exponential, and Polynomial.

Theoretically, all problems belong to exactly one of these three classes, where Exponential is any problem that is not solvable by a polynomial time algorithm.

Unknown Complexity

- **Practically, there are a lot of problems (maybe, most) that have not been proven to be in any of the classes (Yet, maybe never will be).**
- **Most currently "lie between" polynomial and exponential – we know of exponential algorithms, but have been unable to prove that exponential algorithms are necessary.**
- **Some may have polynomial algorithms, but we have not yet been clever enough to discover them.**
- **Linear Programming (real solutions) is $O(n^{3.5})$. That was shown in early 1980s. Prior technique, Simplex, has good observed performance and can be shown polynomial for some subclasses.**

Why do we Care?

If an algorithm is $O(n^k)$, increasing the size of an instance by one gives a running time that is $O((n+1)^k)$

That's really not much more.

With an increase of one in an exponential algorithm, $O(2^n)$ changes to $O(2^{n+1}) = O(2 \cdot 2^n) = 2 \cdot O(2^n)$ – that is, it takes about twice as long.

A Word about “Size”

Technically, the size of an instance is the minimum number of bits (information) needed to represent the instance – its “length.”

This comes from early Formal Language researchers who were analyzing the time needed to ‘recognize’ a string of characters as a function of its length (number of characters).

When dealing with more general problems there is usually a parameter (number of vertices, processors, variables, etc.) that is polynomially related to the length of the instance. Then, we are justified in using the parameter as a measure of the length (size), since anything polynomially related to one will be polynomially related to the other.

The Subtlety of “Size”

But, be careful.

For instance, if the "value" (magnitude) of n is both the input and the parameter, the 'length' of the input (number of bits) is $\log_2(n)$. So, an algorithm that takes n time is running in $n = 2^{\log_2(n)}$ time, which is exponential in terms of the length, $\log_2(n)$, but linear (hence, polynomial) in terms of the "value," or magnitude, of n .

It's a subtle, and usually unimportant difference, but it can bite you.

P = Polynomial Time

- P is the class of decision problems containing all those that can be solved by a deterministic Turing machine using polynomial time in the size of each instance of the problem.
- P contain linear programming over real numbers, but not when the solution is constrained to integers.
- P even contains the problem of determining if a number is prime.

Some Problems in P

- Given $G = (V, E)$ and two vertices $u, v \in V$,
is there a path from u to v ?
Just use depth first search starting at u to determine all vertices reachable from u and see if v is one of them. Can do with undirected or directed graphs. $O(|V| + |E|)$
- Given two positive integers, n, m ,
are n and m relatively prime?
Just run Euclidean algorithm to see if $\text{GCD}(n, m) = 1$.
 $O(\min(\log_2(n), \log_2(m)))$ which is order of the problem representation.
- Given a CFG, $G = (V, \Sigma, S, R)$ and a word $w \in \Sigma^*$,
is w in $L(G)$?
Convert G to CNF and run CKY algorithm, $O(|w|^3)$ or if you are really an algorithm junkie, $O(|w|^{2.3728639})$

NP = Non-Det. Poly Time

- NP is the class of decision problems solvable in polynomial time on a non-deterministic Turing machine.
- Clearly $P \subseteq NP$. Whether or not this is proper inclusion is the well-known challenge $P = NP$?
- NP can also be described as the class of decision problems that can be verified in polynomial time. This is the most useful version of a definition of NP.
- NP can even be described as the class of decision problems that can be solved in polynomial time when no a priori bound is placed on the number of processors that can be used in the algorithm.
- An example is the problem to determine if a boolean expression is satisfiable (more about this later)

NP-Hard

- A is NP-Hard if all NP problems polynomial reduce to A.
- If A is NP-Hard and in NP, then A is NP-Complete.
- QSAT (Quantified SAT) is the problem to determine if an arbitrary fully quantified Boolean expression is true.
Note: SAT only uses existential.
- QSAT is NP-Hard, but may not be in NP.
- QSAT can be solved in polynomial space (PSPACE).

NP-Complete; NP-Hard

- A decision problem, C , is NP-complete if:
 - **C is in NP and**
 - **C is NP-hard. That is, every problem in NP is polynomially reducible to C .**
- D polynomially reduces to C means that there is a deterministic polynomial-time many-one algorithm, f , that transforms each instance x of D into an instance $f(x)$ of C , such that the answer to $f(x)$ is YES if and only if the answer to x is YES.
- To prove that an NP problem A is NP-complete, it is sufficient to show that an already known NP-complete problem polynomially reduces to A . By transitivity, this shows that A is NP-hard.
- A consequence of this definition is that if we had a polynomial time algorithm for any NP-complete problem C , we could solve all problems in NP in polynomial time. That is, $P = NP$.
- Note that NP-hard does not necessarily mean NP-complete, as a given NP-hard problem could be outside NP.

P = NP?

If $P = NP$ then all problems in NP are polynomial problems.

If $P \neq NP$ then all NP-C problems are exponential.

Why should $P = NP$?

Why should P equal NP ?

- There seems to be a huge "gap" between the known problems in P and Exponential. That is, almost all known polynomial problems are no worse than n^3 or n^4 .
- Where are the $O(n^{50})$ problems?? $O(n^{100})$? Maybe they are the ones in NP -Complete?
- It's awfully hard to envision a problem that would require n^{100} , but surely they exist?
- Some of the problems in NP -C just look like we should be able to find a polynomial solution (looks can be deceiving, though).

Why Might $P \neq NP$?

Why should P not equal NP ?

- $P = NP$ would mean, for any problem in NP , that it is just as easy to solve an instance from "scratch," as it is to verify the answer if someone gives it to you. That seems a bit hard to believe.
- There simply are a lot of awfully hard looking problems in NP -Complete (and Co - NP -Complete) and some just don't seem to be solvable in polynomial time.
- Many very smart people have tried for a long time to find polynomial algorithms for some of the problems in NP -Complete - with no luck.

Satisfiability

$U = \{u_1, u_2, \dots, u_n\}$, Boolean variables.

(CNF – Conjunctive Normal Form)

$C = \{c_1, c_2, \dots, c_m\}$,

conjunction (and-ing) of "OR clauses"

Example clause:

$$c_i = (u_4 \vee u_{35} \vee \sim u_{18} \vee u_{3\dots} \vee \sim u_6)$$

SAT

- SAT is the problem to decide of an arbitrary Boolean formula (wff in the propositional calculus) whether or not this formula is satisfiable (has a set of variable assignments that evaluate the expression to true).
- SAT clearly can be solved in time $k2^n$, where k is the length of the formula and n is the number of variables in the formula.
- What we can show is that SAT is NP-complete, providing us our first concrete example of an NP-complete decision problem.

The Proof Idea

- An NDTM M accepts w if and only if, run on w , one of its nondeterministic branches becomes an accepting computation history.
- An accepting computation history is a sequence of configurations where:
 - The first configuration is the initial configuration of M on w .
 - Every subsequent configuration is yielded by the previous configuration – that is, it's a legal move for M .
 - The final configuration is an accepting configuration - that is, its state is q_{ACCEPT} .
- We can use Boolean logical formulas easily to require the first and last of a configuration history, and the middle one with a bit of thought. However, first we need to represent the configuration history in the first place.

Simulating NDTM

- Given a NDTM, M , and an input w , we need to create a formula, $\varphi_{M,w}$, containing a polynomial number of terms that is satisfiable just in case M accepts w in polynomial time.
- The formula must encode within its terms a trace of configurations that includes
 - A term for the starting configuration of the TM
 - Terms for all accepting configurations of the TM
 - Terms that ensure the consistency of each configuration
 - Terms that ensure that each configuration after the first follows from the prior configuration by a single move

Tableaus

A **tableau** is an array of tape alphabet symbols.

It represents a configuration history of **one branch** of our NDTM's nondeterminism.

If the NDTM runs in n^k time, the tableau is an $(n^k \times n^k)$ tableau.

It's big enough downward because, well, the TM runs in n^k .

...and rightward because the TM can only *count* to n^k .

We assume that every configuration starts and ends with a # symbol.

We think of our tableau as looking like this in the "beginning": the starting configuration across the top, and the other configurations blank.

(We quote "beginning" because SAT isn't really a stateful algorithm, but just go with it for now.)

But we've assumed that we can "represent" alphabet symbols. How do we do that, in SAT?

#	q_0	w_1	w_2	...	w_n	□	...	□	#	$\uparrow n^k \downarrow$
#									#	
#									#	
#									#	
#									#	
#									#	
#									#	
#									#	
#									#	
#									#	
$\leftarrow n^k \rightarrow$										

Encoding the Tableau: Basics

Consider a set comprised of:

The tape alphabet

The state set

The separator character

$$C = \Gamma \cup Q \cup \{ \# \}$$

Consider a cell variable:

$$X_{i,j,c}$$

Turning this variable on corresponds to setting cell $(i, j) = c$, for some $c \in C$.

	1	2	3	4	5	6	7	8	9	10
1	#	q_0	w_1	w_2	...	w_n	□	...	□	#
2	#									#
3	#									#
4	#									#
5	#									#
6	#									#
7	#									#
8	#									#
9	#									#
10	#									#

Encoding the Tableau: Cells

Consider our tableau alphabet:

$$C = \Gamma \cup Q \cup \{ \# \}$$

Consider a cell and corresponding variable:

$$x_{i,j,c}$$

Now we need to make sure the tableau is consistently encoded.

Create a clause for **each cell (i, j)**.

$$\phi_{\text{encode}}(i, j) = \left[\left(\bigvee_{c \in C} x_{i,j,c} \right) \wedge \left(\bigwedge_{\substack{c, d \in C \\ c \neq d}} (\overline{x_{i,j,c}} \vee \overline{x_{i,j,d}}) \right) \right]$$

The left demands $x_{i,j,c}$ be true for **some c**.
The right demands $x_{i,j,c}$ be true for **only one c**.

	1	2	3	4	5	6	7	8	9	10
1	#	q_0	w_1	w_2	...	w_n	\square	...	\square	#
2	#									#
3	#									#
4	#									#
5	#									#
6	#									#
7	#									#
8	#									#
9	#									#
10	#									#

Encoding the Tableau: The Tableau

Tableau alphabet: $C = \Gamma \cup Q \cup \{ \# \}$

Cell variable: $x_{i,j,c}$

Create an encoding clause for each cell (i, j) .

$$\phi_{\text{encode}}(i, j) = \left[\left(\bigvee_{c \in C} x_{i,j,c} \right) \wedge \left(\bigwedge_{\substack{c,d \in C \\ c \neq d}} (\overline{x_{i,j,c}} \vee \overline{x_{i,j,d}}) \right) \right]$$

Now repeat the clause across the tableau.

$$\phi_{\text{cells}} = \bigwedge_{1 \leq i, j \leq n^k} \phi_{\text{encode}}(i, j)$$

This is our *cell formula*. It ensures that each cell in the tableau is assigned a single symbol.

	1	2	3	4	5	6	7	8	9	10
1	#	q_0	w_1	w_2	...	w_n	\square	...	\square	#
2	#									#
3	#									#
4	#									#
5	#									#
6	#									#
7	#									#
8	#									#
9	#									#
10	#									#

Encoding the Tableau: Complexity

$$\phi_{\text{encode}}(i, j) = \left[\left(\bigvee_{c \in C} x_{i,j,c} \right) \wedge \left(\bigwedge_{\substack{c,d \in C \\ c \neq d}} (\overline{x_{i,j,c}} \vee \overline{x_{i,j,d}}) \right) \right]$$

We can create the single-cell encoding formula in polynomial time with a $|C|^2$ iteration.

$$\phi_{\text{cells}} = \bigwedge_{1 \leq i, j \leq n^k} \phi_{\text{encode}}(i, j)$$

We can create the *entire* cell formula in polynomial time with an n^{2k} iteration around that.

So we can say that ϕ_{cells} is satisfied by, and only by, a properly encoded tableau, and is created in polynomial time.

	1	2	3	4	5	6	7	8	9	10
1	#	q_0	w_1	w_2	...	w_n	\square	...	\square	#
2	#									#
3	#									#
4	#									#
5	#									#
6	#									#
7	#									#
8	#									#
9	#									#
10	#									#

Starting and Accepting

Starting and accepting are (comparatively) easy.

To start, take the start configuration padded to n^k length with blanks...

$$S = \#q_0w_1w_2\dots w_n\square\dots\square\# \text{ so that } |S| = n^k$$

...and **require the first row be equal to the start configuration**:

$$\phi_{\text{start}} = \bigwedge_{1 \leq j \leq n^k} [x_{1,j,s_j}]$$

Then to accept, just **require an accept state somewhere in the tableau**.

$$\phi_{\text{accept}} = \bigvee_{1 \leq i,j \leq n^k} [x_{i,j,q_A}]$$

	1	2	3	4	5	6	7	8	9	10
1	#	q_0	w_1	w_2	...	w_n	\square	...	\square	#
2	#									#
3	#									#
4	#									#
5	#	w_1	w_2	...	q_A	...	\square	...	\square	#
6	#									#
7	#									#
8	#									#
9	#									#
10	#									#

Starting and Accepting

$$\phi_{\text{start}} = \bigwedge_{1 \leq j \leq n^k} [x_{1,j,s_j}]$$

$$\phi_{\text{accept}} = \bigvee_{1 \leq i,j \leq n^k} [x_{i,j,q_A}]$$

We can generate the start and accept formulas in n^k and $(n^k)^2$ time, both polynomial.

So now we can say that:

ϕ_{start} is satisfied by, and only by, a tableau with the starting configuration of M on w encoded as its first row, and is created in polynomial time.

...and...

ϕ_{accept} is satisfied by, and only by, a tableau encoding an accepting configuration as one of its rows, and is created in polynomial time.

	1	2	3	4	5	6	7	8	9	10
1	#	q_0	w_1	w_2	...	w_n	□	...	□	#
2	#									#
3	#									#
4	#									#
5	#	z_1	z_2	...	q_A	...	□	...	□	#
6	#									#
7	#									#
8	#									#
9	#									#
10	#									#

Transitions

Now, for transitions. Recall the discussions we had about ID changes being limited to three characters or six, when looking at transitions..

A given 2x3 **window** is **legal** if it does not violate our machine's transition function.

Given the linear sets of states and tape symbols, and the finite size of 2x3 windows, we can make a **polynomial-sized set of all legal windows**.

Let a sequence $A = (a_1, \dots, a_6)$ be a 2x3 window, with a_1 the top left cell, a_2 the top middle, etc.

We say that A is **legal** if it represents a legal window. Here we have q_0 a R q_1

	1	2	3	4	5	6	7	8	9	10
1	#	q_0	a	b	c	a	□	□	□	#
2	#	a	q_1	b	c	a	□	□	□	#
3	#									#
4	#									#
5	#									#
6	#									#
7	#									#
8	#									#
9	#									#
10	#									#

Transitions

A given 2x3 **window** is **legal** if it does not violate our machine's transition function. We have a **polynomial-sized set of all legal windows**.

Let a sequence $A = (a_1, \dots, a_6)$ be a 2x3 window. A is **legal** if it **represents a legal window**.

Now we can come up with a formula to say that the window top-centered at cell (i, j) is legal.

$$\phi_{\text{legal}}(i, j) = \bigvee_{\substack{A=(a_1, \dots, a_6) \\ \text{is legal}}} \left[\begin{array}{l} x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge \\ x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6} \end{array} \right]$$

Don't be intimidated by this formula!

It's just **counting off the six cells of the window** and demanding that each be **equal to the corresponding cell in some legal window**.

	1	2	3	4	5	6	7	8	9	10
1	#	q_0	a	b	c	a	□	□	□	#
2	#	a	q_1	b	c	a	□	□	□	#
3	#									#
4	#									#
5	#									#
6	#									#
7	#									#
8	#									#
9	#									#
10	#									#

Transitions

A given 2x3 **window** is **legal** if it does not violate our machine's transition function.

We have a **polynomial-sized set of all legal windows**.

Let a sequence $A = (a_1, \dots, a_6)$ be a 2x3 window. A is **legal** if it represents a legal window.

$$\phi_{\text{legal}}(i, j) = \bigvee_{\substack{A=(a_1, \dots, a_6) \\ \text{is legal}}} \left[\begin{array}{l} x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge \\ x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6} \end{array} \right]$$

Since we have a polynomial number of legal windows, this formula is also polynomial. So we can say:

$\phi_{\text{legal}}(i, j)$ is satisfied by, and only by, a tableau whose window top-centered at (i, j) is legal; and is created in polynomial time.

	1	2	3	4	5	6	7	8	9	10
1	#	q_0	a	b	c	a	□	□	□	#
2	#	a	q_1	b	c	a	□	□	□	#
3	#									#
4	#									#
5	#									#
6	#									#
7	#									#
8	#									#
9	#									#
10	#									#

Windows and Configurations

Consider any **upper** and **lower** configuration in the tableau, so that the lower configuration is the one immediately below – that is, following – the upper.

If all the windows top-centered on cells in the upper configuration are legal, then:

The legality of the windows that don't involve the state symbol easily ensures the legality of the configuration below them.

The window top-centered on the state symbol in the upper configuration is sufficient to ensure that the state symbol in the lower configuration makes a legal move.

The upper configuration yields the lower one if and only if all the windows top-centered on cells in the upper configuration are legal – and that holds all the way down the tableau.

	1	2	3	4	5	6	7	8	9	10
1	#	q_0	a	b	c	a	□	□	□	#
2	#	a	q_1	b	c	a	□	□	□	#
3	#									#
4	#									#
5	#									#
6	#									#
7	#									#
8	#									#
9	#									#
10	#									#

Windows and Configurations

$$\phi_{\text{legal}}(i, j) = \bigvee_{\substack{A=(a_1, \dots, a_6) \\ \text{is legal}}} \left[x_{i, j-1, a_1} \wedge x_{i, j, a_2} \wedge x_{i, j+1, a_3} \wedge x_{i+1, j-1, a_4} \wedge x_{i+1, j, a_5} \wedge x_{i+1, j+1, a_6} \right]$$

$\phi_{\text{legal}}(i, j)$ is satisfied by, and only by, a tableau whose window top-centered at (i, j) is legal; and is created in polynomial time.

An upper configuration yields a lower one iff all the windows top-centered within the upper are legal.

This holds all the way down the tableau.

Then we have:

$$\phi_{\text{move}} = \bigwedge_{\substack{1 \leq i < n^k, \\ 1 < j < n^k}} \phi_{\text{legal}}(i, j)$$

And can say ϕ_{move} is satisfied by, and only by, a tableau that does not violate the machine's transition function; and is created in polynomial time.

	1	2	3	4	5	6	7	8	9	10
1	#	q_0	a	b	c	a	□	□	□	#
2	#	a	q_1	b	c	a	□	□	□	#
3	#									#
4	#									#
5	#									#
6	#									#
7	#									#
8	#									#
9	#									#
10	#									#

Pulling It Together

$$\phi_{\text{cells}} = \bigwedge_{1 \leq i, j \leq n^k} \phi_{\text{encode}}(i, j)$$

$$\phi_{\text{start}} = \bigwedge_{1 \leq j \leq n^k} [x_{1, j, s_j}]$$

$$\phi_{\text{accept}} = \bigvee_{1 \leq i, j \leq n^k} [x_{i, j, q_A}]$$

$$\phi_{\text{move}} = \bigwedge_{\substack{1 \leq i < n^k, \\ 1 < j < n^k}} \phi_{\text{legal}}(i, j)$$

$$\phi_{\text{NDTM}} = (\phi_{\text{cells}} \wedge \phi_{\text{start}} \wedge \phi_{\text{accept}} \wedge \phi_{\text{move}})$$

We have:

ϕ_{cells} is satisfied by, and only by, a properly encoded tableau.

ϕ_{start} is satisfied by, and only by, a tableau with the starting configuration of M on w encoded as its first row.

ϕ_{accept} is satisfied by, and only by, a tableau encoding an accepting configuration as one of its rows.

ϕ_{move} is satisfied by, and only by, a tableau that does not violate the machine's transition function.

All are created in polynomial time.

Then ϕ_{NDTM} is satisfied by, and only by, a **tableau encoding an accepting computation history of M on w , and is created in polynomial time.**

SAT is NP-Complete

$$\phi_{\text{NDTM}} = (\phi_{\text{cells}} \wedge \phi_{\text{start}} \wedge \phi_{\text{accept}} \wedge \phi_{\text{move}})$$

ϕ_{NDTM} created from NDTM M and input w is satisfied by, and only by, a tableau encoding an accepting computation history of M on w , and is created in polynomial time.

This means that:

SAT accepts ϕ_{NDTM} if and only if such a tableau exists...

...if and only if the NDTM we are encoding into ϕ_{NDTM} accepts w .

We've just polynomially reduced every possible NP language to SAT .

Let's convince ourselves of that a bit more.

By definition, any NP language has an NDTM M that decides it in polynomial time.

We can decide any NP language with a result from SAT using the following algorithm:

On input $\langle M, w \rangle$:

Create ϕ_{NDTM} from M and w .

Run the decider for SAT on ϕ_{NDTM} .

Accept if SAT accepts, reject if it rejects.

SAT is NP-complete.

NP-Complete

Within a year, Richard Karp added 22 problems to this special class.

We will focus on:

3-SAT

SubsetSum

Partition

Integer Linear Programming

Vertex Cover

Independent Set

K-Color

Multiprocessor Scheduling

Co-NP

- A problem is in co-NP if its complement is in NP
 - this is like co-RE, wrt RE problems.
- An example is the problem to determine if a Boolean expression is a tautology.
 - If the answer to the problem "is B in TAUT ?" is NO, then $\neg A$ is in SAT.
- A more direct example of a co-NP problem is to determine if a Boolean expression is self-contradictory.
 - This is the complement of satisfiability.
- Both of the above are co-NP Complete

SAT to 3SAT

- 3-SAT means that each clause has exactly three terms
- If one term, e.g., (p) , expand to $(p \vee p \vee p)$
- If two terms, e.g., $(p \vee q)$, expand to $(p \vee q \vee p)$
- Any clause with three terms is fine
- If $n > 3$ terms, can reduce to two clauses, one with three terms and one with $n-1$ terms, e.g., $(p_1 \vee p_2 \vee \dots \vee p_n)$ to $(p_1 \vee p_2 \vee z)$ & $(p_3 \vee \dots \vee p_n \vee \sim z)$, where z is a new variable. If $n=4$, we are done, else apply this approach again with the clause having $n-1$ terms

SubsetSum

$$S = \{s_1, s_2, \dots, s_n\}$$

set of positive integers
and an integer B.

**Question: Does S have a subset whose
values sum to B?**

No one knows of a polynomial algorithm.

{No one has proven there isn't one, either!!}

SubsetSum \equiv_p Partition

Theorem. SAT \leq_p 3SAT

Theorem. 3SAT \leq_p SubsetSum

Theorem. SubsetSum \leq_p Partition

Theorem. Partition \leq_p SubsetSum

Therefore, not only is Satisfiability in NP-Complete, but so is 3SAT, Partition, and SubsetSum.

3SAT \leq_p SubsetSum

Assuming a 3SAT expression $(a + \sim b + c) (\sim a + b + \sim c)$

	a	b	c	$a+\sim b+c$	$\sim a+b+\sim c$
a	1	0	0	1	0
$\sim a$	1	0	0	0	1
b	0	1	0	0	1
$\sim b$	0	1	0	1	0
c	0	0	1	1	0
$\sim c$	0	0	1	0	1
C1	0	0	0	1	0
C1'	0	0	0	1	0
C2	0	0	0	0	1
C2'	0	0	0	0	1
	1	1	1	3	3

SubsetSum \equiv_p Partition Details

- Partition is polynomial equivalent to SubsetSum
 - Let i_1, i_2, \dots, i_n, G be an instance of SubsetSum. This instance has answer “yes” iff $i_1, i_2, \dots, i_n, 2 \cdot \text{Sum}(i_1, i_2, \dots, i_n) - G, \text{Sum}(i_1, i_2, \dots, i_n) + G$ has answer “yes” in Partition. Here we assume that $G \leq \text{Sum}(i_1, i_2, \dots, i_n)$, for, if not, the answer is “no.”
 - Let i_1, i_2, \dots, i_n be an instance of Partition. This instance has answer “yes” iff $i_1, i_2, \dots, i_n, \text{Sum}(i_1, i_2, \dots, i_n)/2$ has answer “yes” in SubsetSum

SubsetSum \equiv_p Partition

- [(15, 17, 27, 11, 4, 12, 33, 5, 6, 21, 2), 57]
- A solution is 15, 17, 11, 12, 2
- Sum of all is 153
- Mapping to Partition is
 - (15, 17, 27, 11, 4, 12, 33, 5, 6, 21, 2, 306-57, 153+57)
 - (15, 17, 27, 11, 4, 12, 33, 5, 6, 21, 2, 249, 210)
 - (15+17+11+12+2+249) = 306
 - (27+4+33+5+6+21+210) = 306
- Going other direction map above to
 - [(15, 17, 27, 11, 4, 12, 33, 5, 6, 21, 2, 249, 210), 306]

Integer Linear Programming

- Show for 0-1 integer linear programming by constraining solution space. Start with an instance of SAT (or 3SAT), assuming variables v_1, \dots, v_n and clauses c_1, \dots, c_m
- For each variable v_i , have constraint that $0 \leq v_i \leq 1$
- For each clause we provide a constraint that it must be satisfied (evaluate to at least 1). For example, if clause c_j is $v_2 \vee \sim v_3 \vee v_5 \vee v_6$ then add the constraint $v_2 + (1-v_3) + v_5 + v_6 \geq 1$
- A solution to this set of integer linear constraints implies a solution to the instance of SAT and vice versa

Assignment # 10

1. Recast the decision problem for the Boolean expression $(p+q+\sim r)(p+\sim q)(r)$ as a SubsetSum problem using the construction discussed in class. Indicate what rows would need to be chosen for a solution.
2. Recast the SubsetSum problem $\{7, 17, 4, 11, 6, 2, 7\}$, $G=36$ as a Partition Problem using the construction from class. Indicate what values need to be chosen to equal 36 for the SubsetSum problem. Indicate the partitions that evenly divide the Partition Problem you posed.
3. Recast the decision problem for the Boolean expression $(p+q+\sim r)(p+\sim q)(r)$ as a 0,1-Integer Linear Programming problem using the construction discussed in class. Indicate what binary (0,1) values of p , q , and r give rise to a solution to the Integer Linear Programming problem you posed.

Due: Tuesday, Dec. 3, 7:00 PM (use Webcourses). I will post answers when I get home after class so you cannot miss the deadline.

VERTEX COVERING (VC) DECISION PROBLEM IS NP-HARD

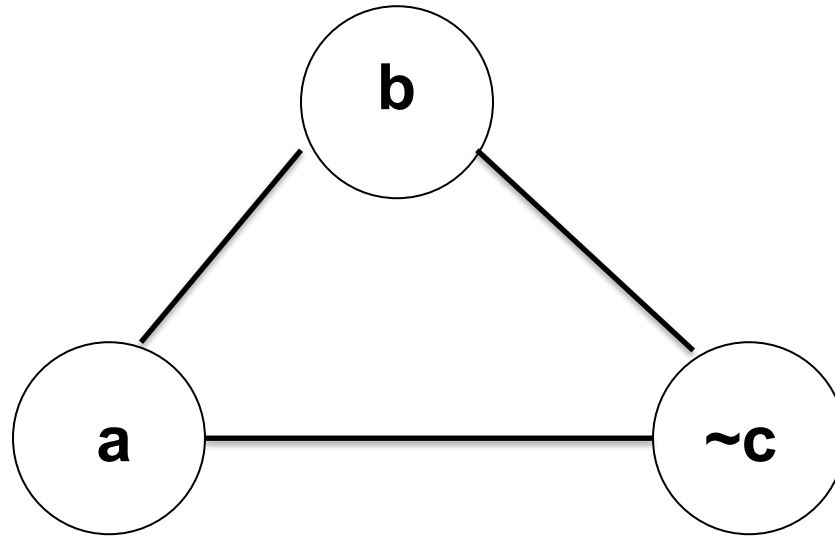
3SAT to Vertex Cover

- **Vertex cover seeks a set of vertices that cover every edge in some graph**
- **Let $I_{3\text{-SAT}}$ be an arbitrary instance of 3-SAT. For integers n and m , $U = \{u_1, u_2, \dots, u_n\}$ and $C_i = \{z_{i1}, z_{i2}, z_{i3}\}$ for $1 \leq i \leq m$, where each z_{ij} is either a u_k or u_k' for some k .**
- **Construct an instance of VC as follows.**
- **For each i , $1 \leq i \leq n$, construct two vertices, u_i and u_i' with an edge between them.**
- **For each clause $C_i = \{z_{i1}, z_{i2}, z_{i3}\}$, $1 \leq i \leq m$, construct three vertices z_{i1} , z_{i2} , and z_{i3} and form a "triangle on them. Each z_{ij} is one of the Boolean variables u_k or its complement u_k' . Draw an edge between z_{ij} and the Boolean variable (whichever it is). Each z_{ij} has degree 3. Finally, set $k = n+2m$.**
- **Theorem. The given instance of 3-SAT is satisfiable if and only if the constructed instance of VC has a vertex cover with at most k vertices.**

VC Variable Gadget



VC Clause Gadget

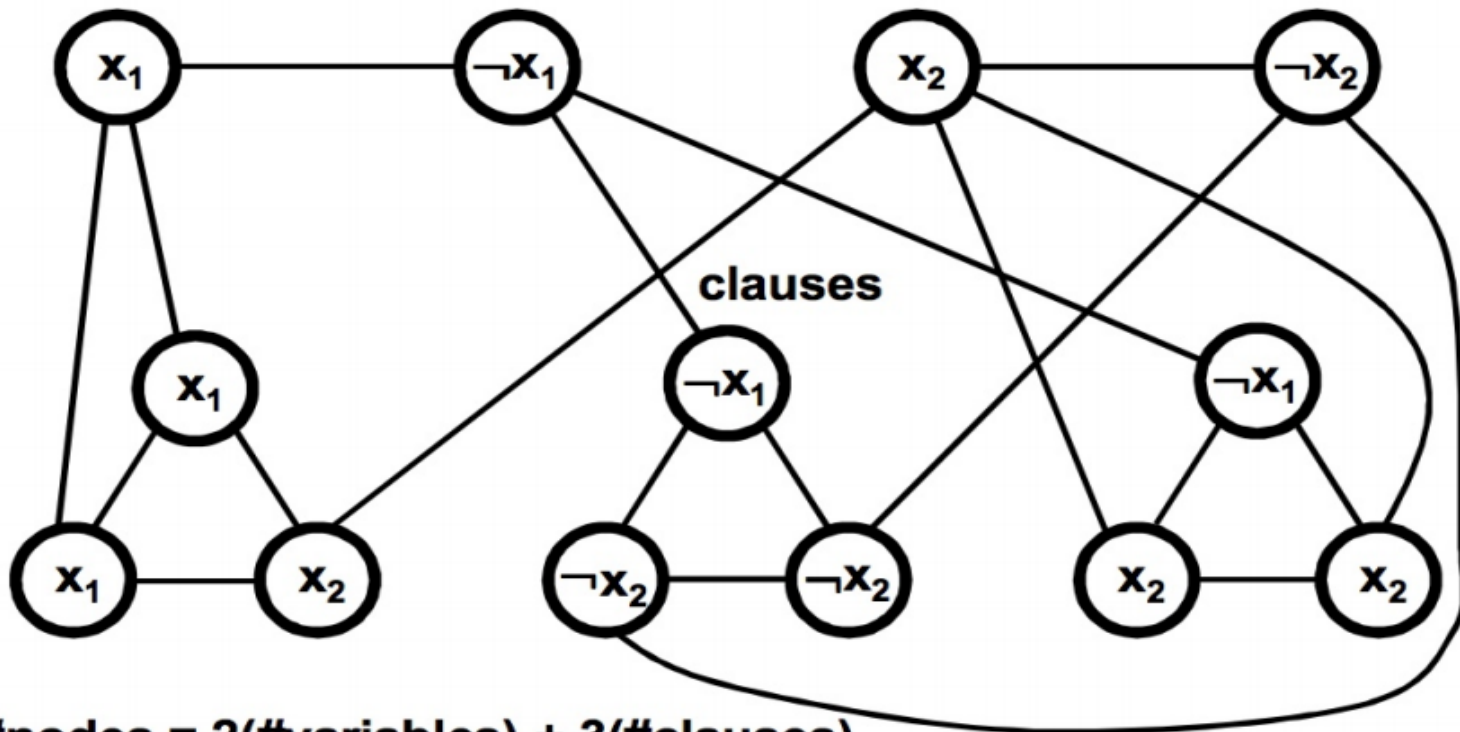


$$a + b + \sim c$$

VC Gadgets Combined

$$(x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$$

Variables and negations of variables



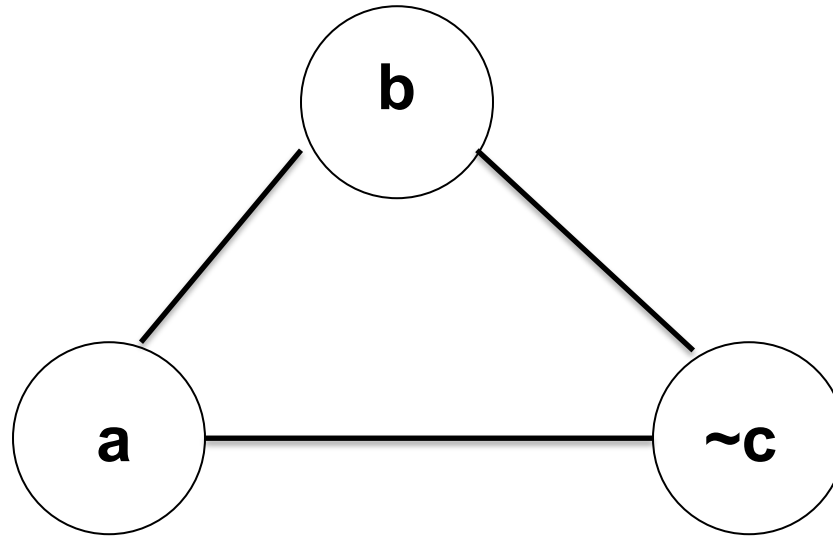
$$\#nodes = 2(\#variables) + 3(\#clauses)$$

Choose a cover that involves $n(2) + 2m(6)$ nodes

Independent Set

- Independent Set
 - Given Graph $G = (V, E)$, a subset S of the vertices is independent if there are no edges between vertices in S
 - The k -IS problem is to determine for a $k > 0$ and a graph G , whether or not G has an independent set of k nodes
- Note there is a related NP-Hard optimization problem to find a Maximum Independent Set. It is even hard to approximate a solution to the Maximum Independent Set Problem.

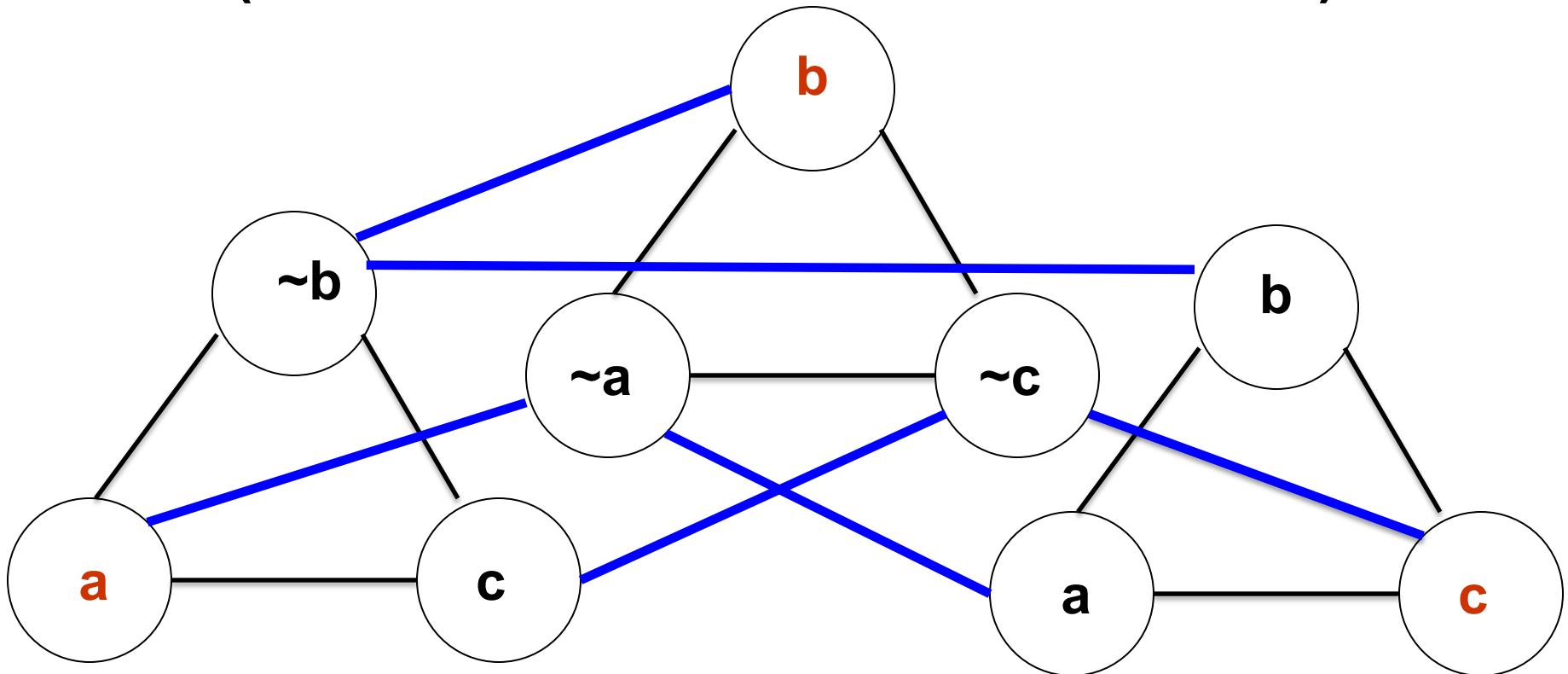
IS (VC) Clause Gadget



$$a + b + \sim c$$

3SAT to IS

$(a + \sim b + c) (\sim a + b + \sim c)(a + b + c)$, $k=3$
(k =number of clauses, not variables)



K-COLOR (KC) DECISION PROBLEM IS NP-HARD

K-Coloring

Given:

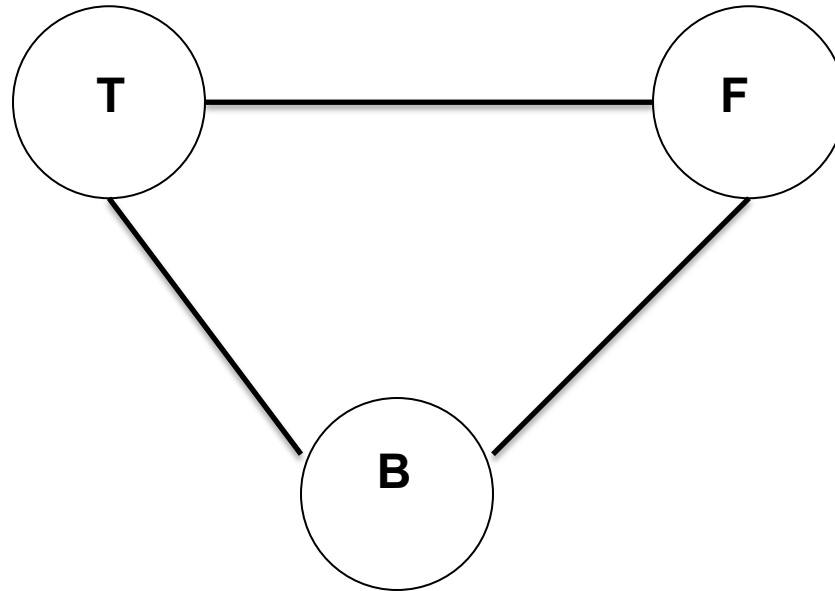
A graph $G = (V, E)$ and an integer k .

Question:

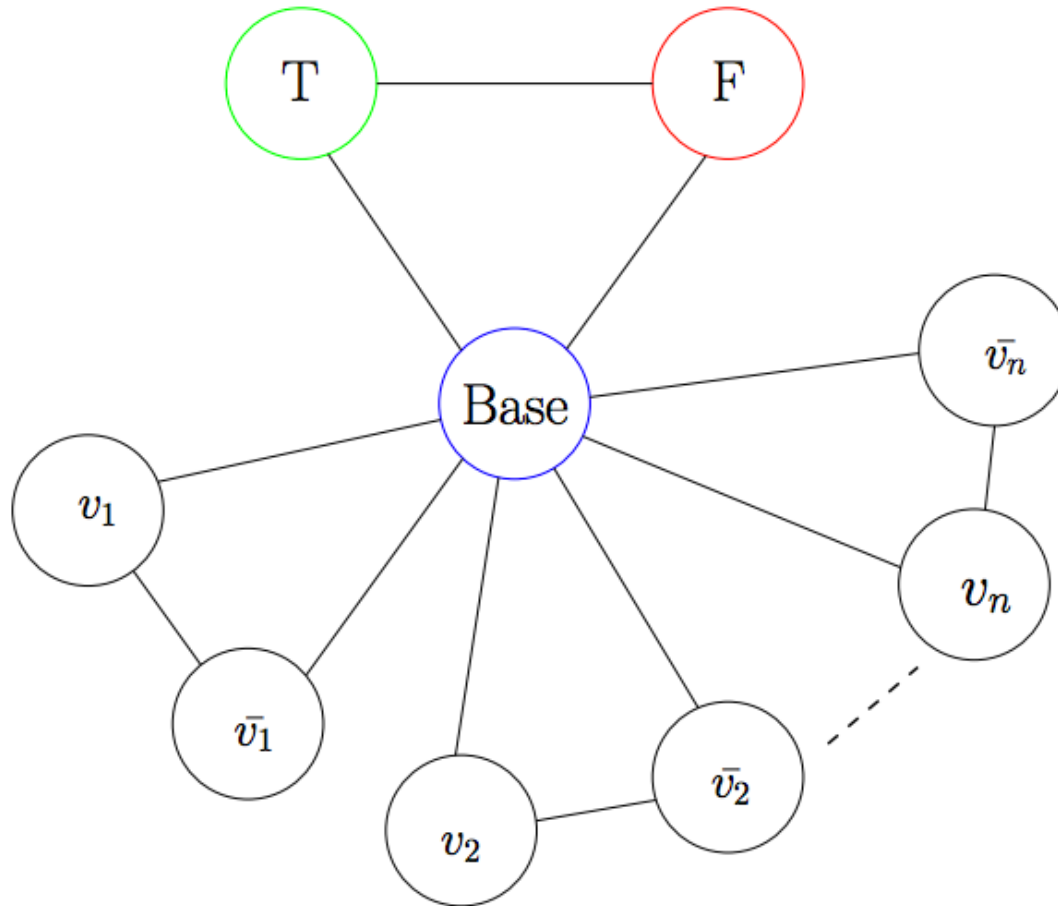
Can the vertices of G be assigned colors from a palette of size k , so that adjacent vertices have different colors and use at most k colors?

3Coloring (3C) uses $k=3$

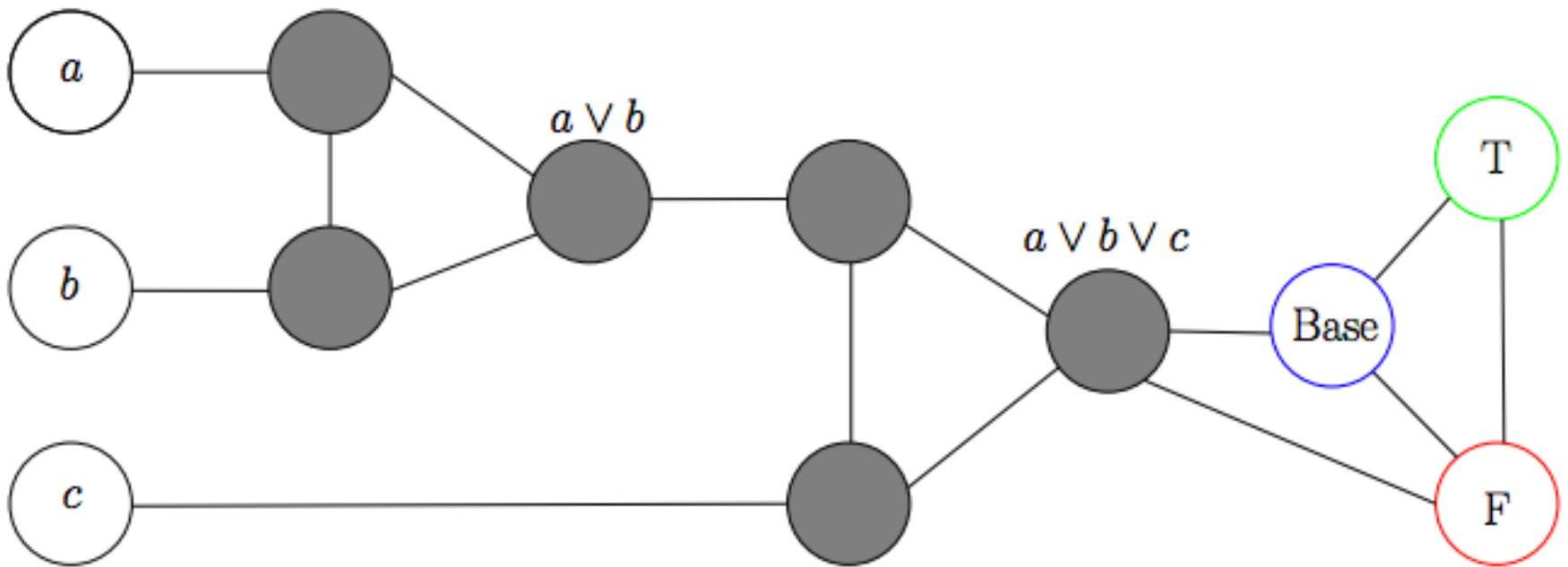
3C Super Gadget



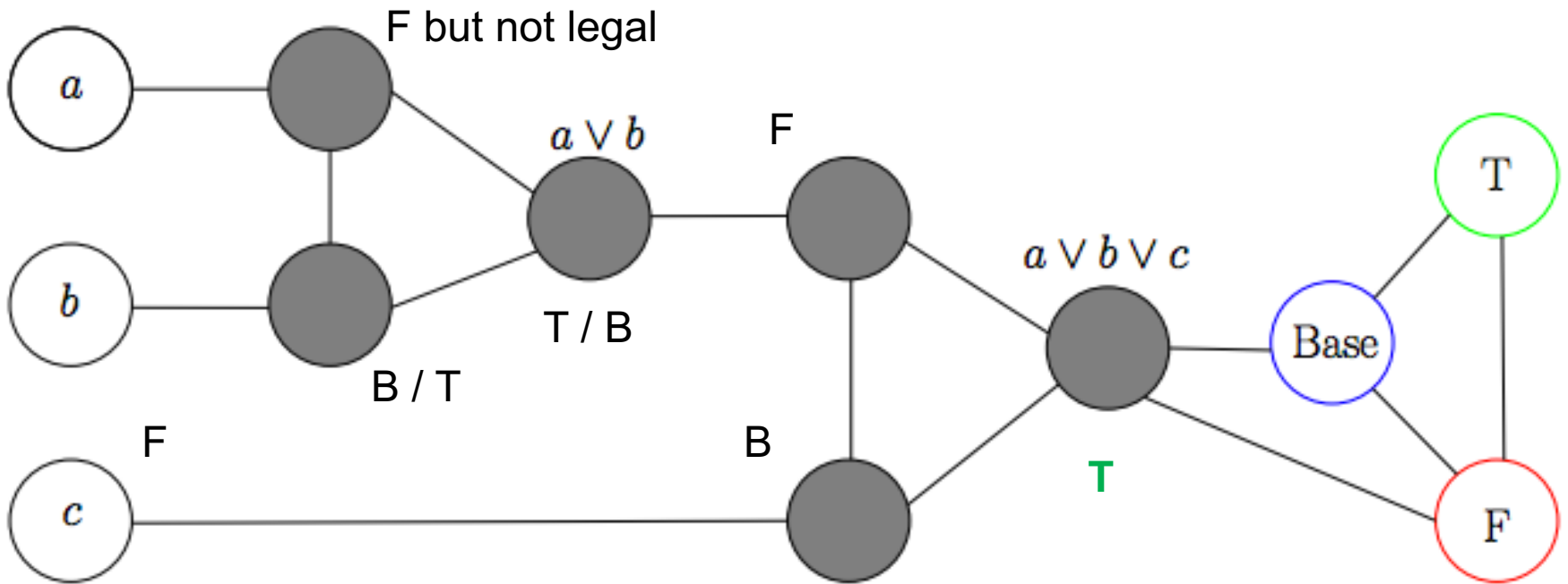
KC Super + Variables Gadget



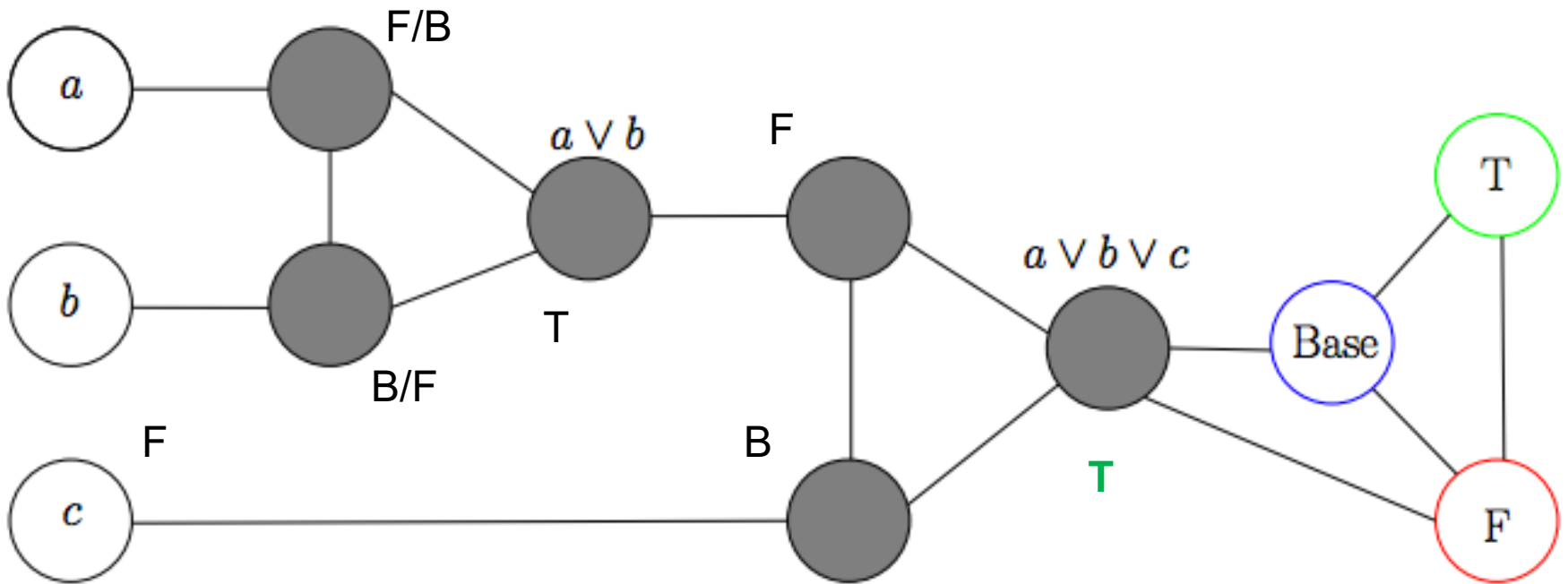
KC Clause Gadget



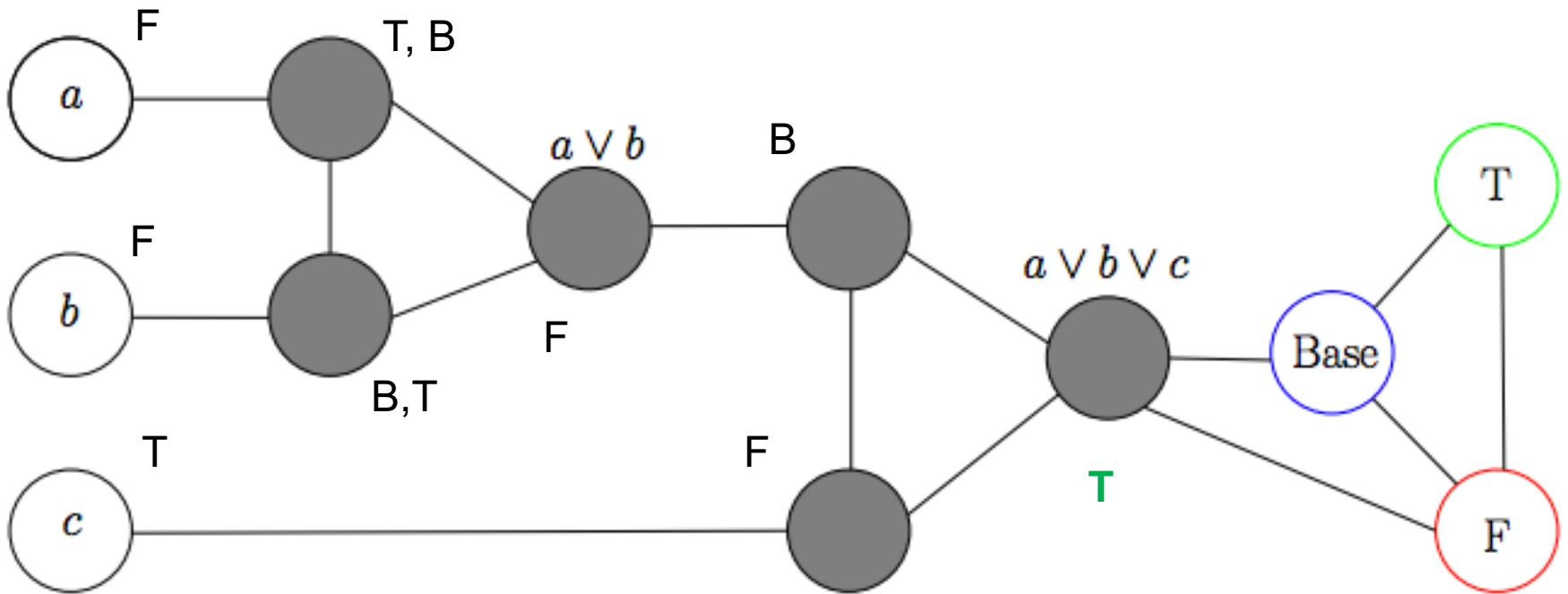
Consider $\sim a, \sim b, \sim c$



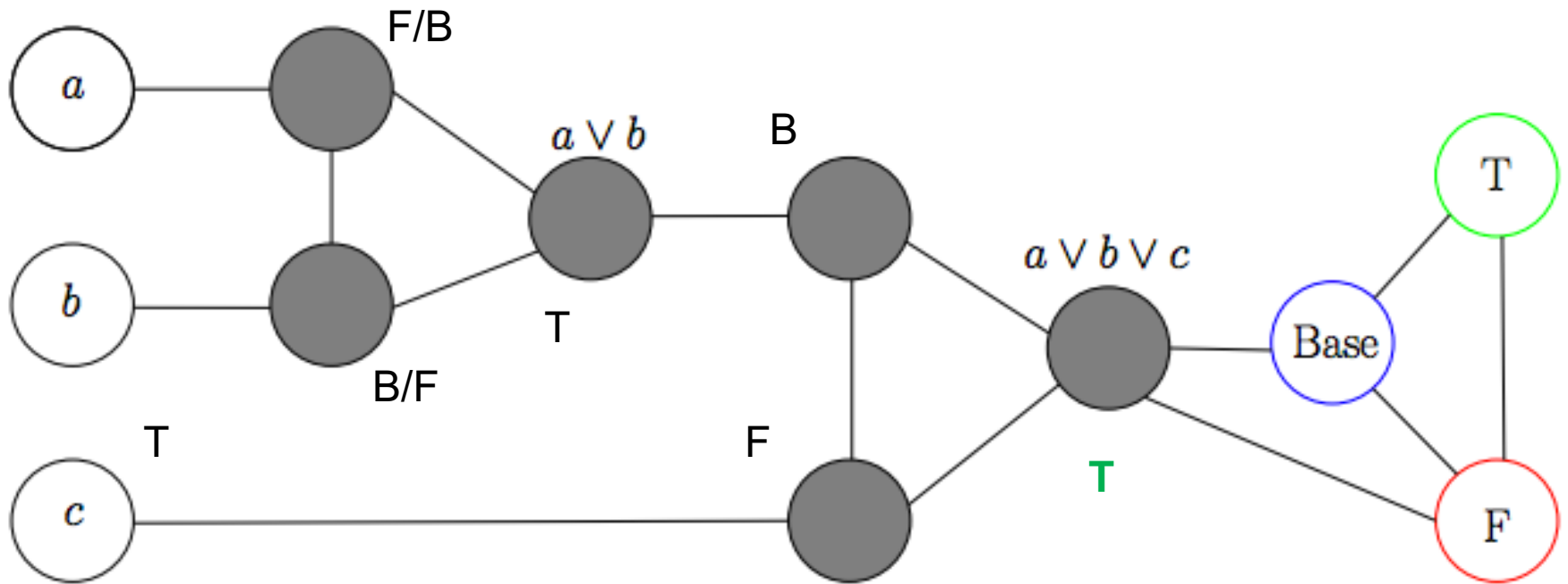
Consider $a \parallel b, \sim c$



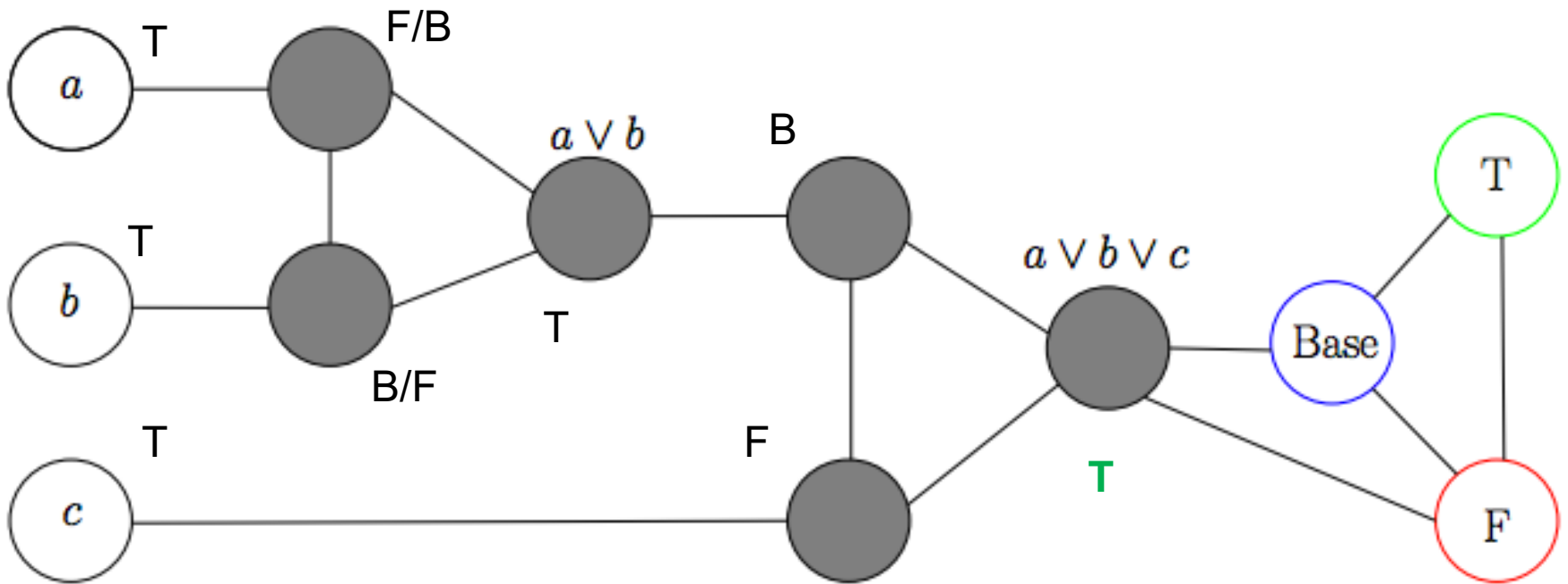
Consider $\sim a, \sim b, c$



Consider one of $a \parallel b, c$

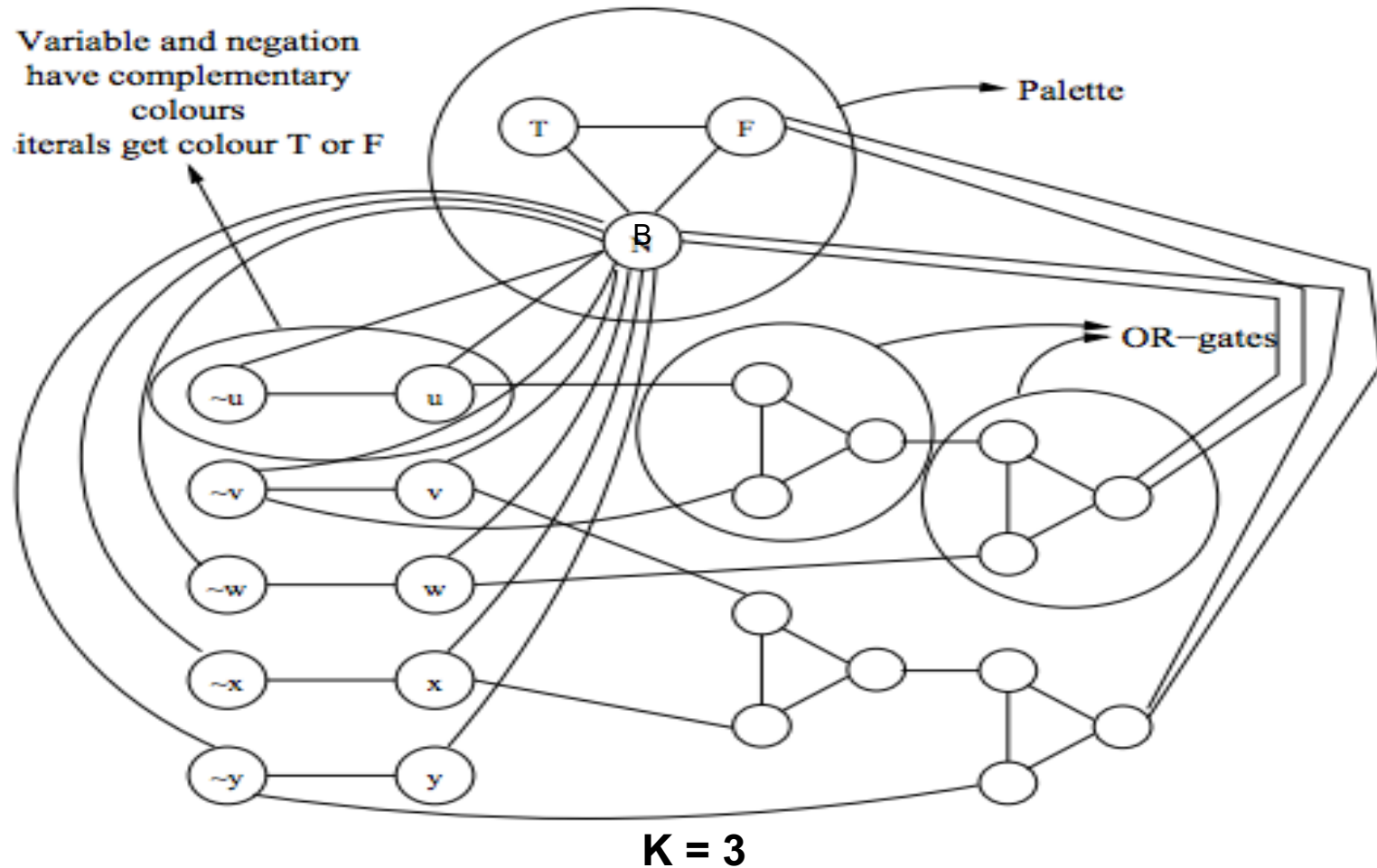


Consider a, b, c



KC Gadgets Combined

$$(u + \sim v + w) (v + x + \sim y)$$



Register Allocation

- **Liveness: A variable is live if its current assignment may be used at some future point in a program's flow**
- **Optimizers often try to keep live variables in registers**
- **If two variables are simultaneously live, they need to be kept in separate registers**
- **Consider the K-coloring problem (can the nodes of a graph be colored with at most K colors under the constraint that adjacent nodes must have different colors?)**
- **Register Allocation reduces to K-coloring by mapping each variable to a node and inserting an edge between variables that are simultaneously live**
- **K-coloring reduces to Register Allocation by interpreting nodes as variables and edges as indicating concurrent liveness**
- **This is a simple mapping because it's an isomorphism**

PROCESSOR SCHEDULING IS NP-HARD

Processor Scheduling

- A Process Scheduling Problem can be described by
 - m processors P_1, P_2, \dots, P_m ,
 - processor timing functions S_1, S_2, \dots, S_m , each describing how the corresponding processor responds to an execution profile,
 - additional resources R_1, R_2, \dots, R_k , e.g., memory
 - transmission cost matrix C_{ij} ($1 \leq i, j \leq m$), based on proc. data sharing,
 - tasks to be executed T_1, T_2, \dots, T_n ,
 - task execution profiles A_1, A_2, \dots, A_n ,
 - a partial order defined on the tasks such that $T_i < T_j$ means that T_i must complete before T_j can start execution,
 - communication matrix D_{ij} ($1 \leq i, j \leq n$); D_{ij} can be non-zero only if $T_i < T_j$,
 - weights W_1, W_2, \dots, W_n -- cost of deferring execution of task.

Complexity Overview

- **The intent of a scheduling algorithm is to minimize the sum of the weighted completion times of all tasks, while obeying the constraints of the task system. Weights can be made large to impose deadlines.**
- **The general scheduling problem is quite complex, but even simpler instances, where the processors are uniform, there are no additional resources, there is no data transmission, the execution profile is just processor time and the weights are uniform, are very hard.**
- **In fact, if we just specify the time to complete each task and we have no partial ordering, then finding an optimal schedule on two processors is an NP-complete problem. It is essentially the subset-sum problem.**

2 Processor Scheduling

The problem of optimally scheduling n tasks T_1, T_2, \dots, T_n onto 2 processors with an empty partial order $<$ is the same as that of dividing a set of positive whole numbers into two subsets, such that the numbers are as close to evenly divided. So, for example, given the numbers

3, 2, 4, 1

we could try a “greedy” approach as follows:

put 3 in set 1

put 2 in set 2

put 4 in set 2 (total is now 6)

put 1 in set 1 (total is now 4)

This is not the best solution. A better option is to put 3 and 2 in one set and 4 and 1 in the other. Such a solution would have been attained if we did a greedy solution on a sorted version of the original numbers. In general, however, sorting doesn't work.

2 Processor Nastiness

Try the unsorted list

7, 7, 6, 6, 5, 4, 4, 5, 4

Greedy (Always in one that is least used)

7, 6, 5, 5 = 23

7, 6, 4, 4, 4 = 25

Optimal

7, 6, 6, 5 = 24

7, 4, 4, 4, 5 = 24

Sort it

7, 7, 6, 6, 5, 5, 4, 4, 4

7, 6, 5, 4, 4 = 26

7, 6, 5, 4 = 22

Even worse than greedy unsorted !!

Heuristics

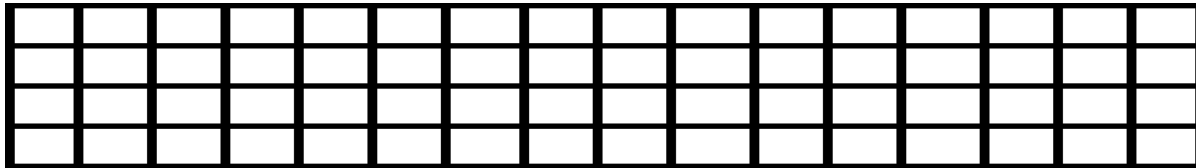
While it is not known whether or not $P = NP?$, it is clear that we need to “solve” problems that are NP-complete since many practical scheduling and networking problems are in this class. For this reason we often choose to find good “heuristics” which are fast and provide acceptable, though not perfect, answers. The First Fit and Best Fit algorithms we previously discussed are examples of such acceptable, imperfect solutions.

Challenge Problem

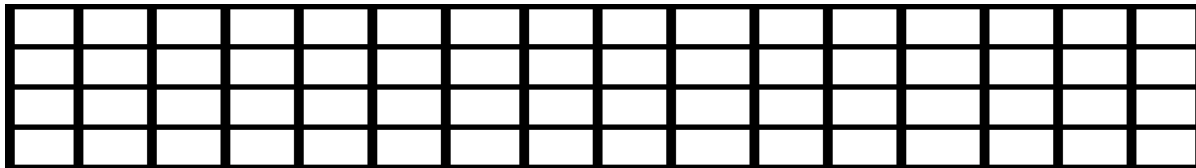
Consider the simple scheduling problem where we have a set of independent tasks running on a fixed number of processors, and we wish to minimize finishing time.

How would a list (first fit, no preemption) strategy schedule tasks with the following IDs and execution times onto four processors? Answer using Gantt chart.

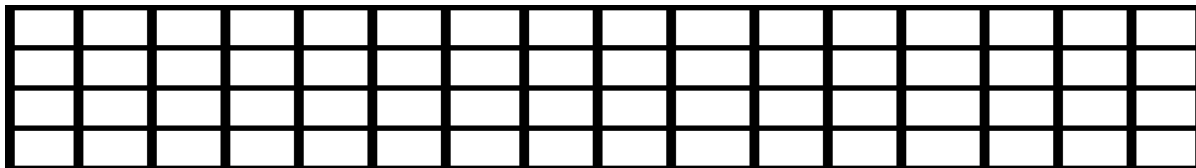
(T1,4) (T2,1) (T3,3) (T4,6) (T5,2) (T6,1) (T7,4) (T8,5) (T9,7) (T10,3) (T11,4) **(2-1/m)**



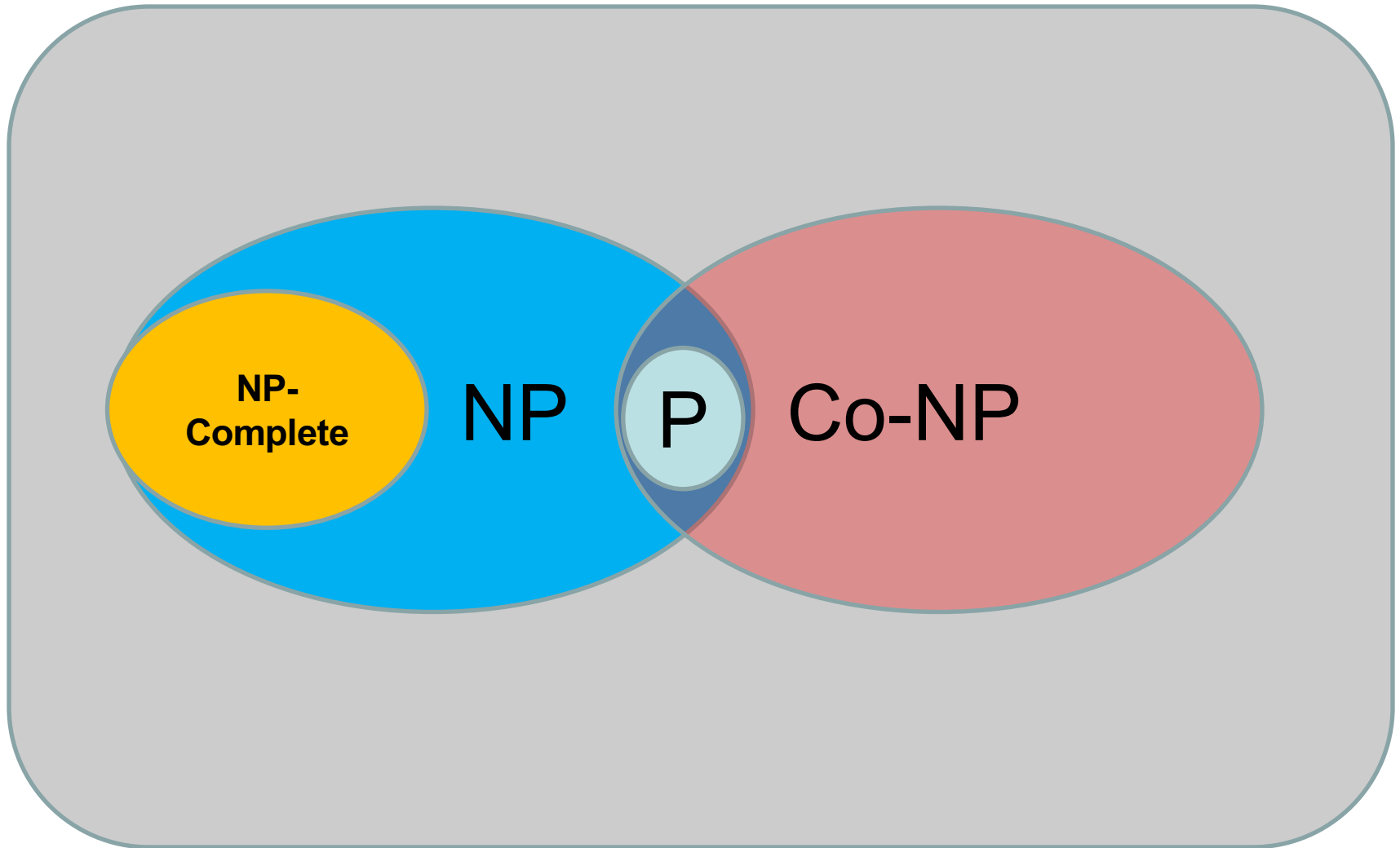
Now show what would happen if the times were sorted non-decreasing. **(2-1/m)**



Now show what would happen if the times were sorted non-increasing. **(4/3-1/3m)**



UNIVERSE OF SETS



Final Exam Topics 1

- Regular languages
 - Decision Problems
 - Membership
 - Emptiness
 - Finiteness
 - Σ^*
 - Equality
 - Containment
 - Closure
 - Union/Concatenation/Star
 - Complement
 - Substitution/Quotient, Prefix, Infix, Suffix
 - Max/Min

Final Exam Topics 2

- Context free languages
 - Writing a simple CFG
 - Decision Problems
 - Membership (CKY)
 - Emptiness (Reduced grammar)
 - Finiteness (Reduced grammar)
 - Σ^* (undecidable)
 - Equality (undecidable)
 - Containment (undecidable)
 - Closure
 - Union/Concatenation/Star
 - Intersection with Regular
 - Substitution/Quotient with Regular, Prefix, Infix, Suffix
 - Non-closure
 - intersection, complement, quotient, Max/Min
 - Pumping Lemma for CFLs

Final Exam Topics 3

- Chomsky Hierarchy
(Red involve no constructive questions)
 - Regular, CFG, CSG, **PSG** (type 3 to type 0)
 - **FSA**s, **PDA**s, **LBA**s, **Turing machines**
 - Length preservation or increase makes membership in associated languages decidable for all but PSGs
 - CFLs can be inherently ambiguous but that does not mean a language that has an ambiguous grammar is automatically inherently ambiguous

Final Exam Topics 4

- Computability Theory
 - Decision problems: solvable (decidable, recursive), semi-decidable (recognizable, recursively enumerable/re, generable), non-re
 - A set is re iff it is semi-decidable
 - If set is re and complement is also re, set is recursive (decidable)
 - Halting problem (K_0): diagonalization proof of undecidability
 - Set K_0 is re but complement is not
 - Set $K = \{ f \mid f(f) \text{ converges} \}$
 - Algorithms (Total): diagonalization proof of non-re
 - Reducibility to show certain problems are not decidable or even non-re
 - K and K_0 are re-complete – reducibility to show these results
 - Rice's Theorem: All non-trivial I/O properties of functions are undecidable (weak and strong versions)
 - Use of quantification to discover upper bound on complexity

Final Exam Topics 5

- Computability Applied to Formal Grammars
(Red only results not constructions that lead to these)
 - Post Correspondence problem (PCP)
 - Definition
 - Undecidability (proof was only sketched and is not part of this course)
 - Application to ambiguity and non-emptiness of intersections of CFLs and to non-emptiness of CSLs
 - Traces of Turing computations
 - Not CFLs
 - Single steps are CFLs (use reversal of second configuration)
 - Intersections of pairwise correct traces are traces
 - Complement of traces (including terminating traces) are CFL
 - Use to show cannot decide if CFL, L , is Σ^*
 - $L = \Sigma^*$ and $L = L^2$ are undecidable for CFLs
 - PSG can mimic TM, so generate any re language; thus, membership in PSL is undecidable, as is emptiness of PSL.
 - All re sets are homomorphic images of CSLs (erase fill character)

Final Exam Topics 6

- Complexity Theory
 - Verifiers versus solvers: P versus NP
 - Definitions of NP: verify in deterministic poly time vs solve in non-deterministic polynomial time
 - Co-P and co-NP; NP-Hard versus NP-Complete
 - Basic idea behind SAT as NP-Complete
 - Reduction of SAT to 3-SAT to Subset-Sum
 - Equivalence of Subset-Sum to Partition
 - Relation of Subset-Sum and Partition to multiprocessor scheduling
 - Vertex cover, 3-coloring, register allocation, Independent set, 0-1 Integer Linear Programming
 - Gadgets for above