Who, What, Where and When

• Instructor: Charles Hughes;
  HEC-247C
  charles.hughes@ucf.edu
  (e-mail is a good way to get me)
  Use Subject: COT6410
  Office Hours: TR 3:15PM-4:30PM

• Web Page: http://www.cs.ucf.edu/courses/cot6410/Spring2019

• Meetings: TR 1:30PM-2:45PM, HEC-103;
  28 periods, each 75 minutes long.
  Final Exam (Tuesday, April 30 from 1:00PM to 3:50PM) is
  separate from class meetings

• GTA: Harish Raviprakash; harishr@knights.ucf.edu
  Use Subject: COT6410
  Office Hours: MW 1:30PM-3:00PM; Room: HEC-308
Text Material

• References:
• Draft available at http://www.wisdom.weizmann.ac.il/~oded/cc-drafts.html
• Draft available at http://www.wisdom.weizmann.ac.il/~oded/bc-drafts.html
Goals of Course

• Introduce Computability and Complexity Theory, including
  – Review background on automata and formal languages
  – Basic notions in theory of computation
    • Algorithms and effective procedures
    • Decision and optimization problems
    • Decision problems have yes/no answer to each instance
  – Limits of computation
    • Turing Machines and other equivalent models
    • Determinism and non-determinism
    • Undecidable problems
    • The technique of reducibility; The ubiquity of undecidability (Rice’s Theorem)
    • The notions of semi-decidable (re) and of co-re sets
  – Complexity theory
    • Order notation (quick review)
    • Polynomial reducibility
    • Time complexity, the sets P, NP, co-NP, NP-complete, NP-hard, etc., and the question does P=NP? Sets in NP and NP-Complete.
    • Gadgets and other reduction techniques
Expected Outcomes

• You will gain a solid understanding of various types of computational models and their relations to one another.
• 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, co-NP, NP-complete and NP-Hard.
• You will (hopefully) come away with stronger formal proof skills and a better appreciation 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 do, however, ask lots of questions in class and give lots of 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 notes 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 assignments is encouraged.

• Late Assignments
  – I will accept no late assignments, except under very unusual conditions, and those exceptions must be arranged with me 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.
Grading

• Grading of Assignments and Exams
  – I will endeavor to return each exam within a week of its taking place and each assignment within a week of its due date.

• Exam Weights
  – The weights of exams will be adjusted to your personal benefits, as I weigh the exam you do well in more than one in which you do less well.
Important Dates

• Midterm – Tues., March 5 (tentative)
• Spring Break – March 11-16
• Withdraw Deadline – Wednesday, March 20
• Final – Tues., April 30, 1:00PM–3:50PM
Evaluation (tentative)

• Mid Term – 125 points ; Final – 200 points
• Assignments – 75 points; Paper and Presentation – 75 points
• Extra – 25 points used to increase weight of exams or maybe paper/presentation, always to your benefit
• Total Available: 500 points
• Grading will be  A >= 90%, B+ >= 85%, B >= 80%, C+ >= 75%, C >= 70%, D >= 50%, F < 50% (Minuses might be used)
Decision Problems

• A set of input data items (input "instances" or domain)
• Each input data item defines a question with an answer Yes/No or True/False or 1/0.
• A decision problem can be viewed as a relation between its domain and its binary range
• A decision problem can also be viewed as a partition of the input domain into those that give rise to true instances and those that give rise to false instances.
• In each case, we seek an algorithmic solution (in the form of a predicate) or a proof that none exists
• When an algorithmic solution exists, we seek an efficient algorithm, or proofs of the problem’s inherent complexity
UNIVERSE OF DISCOURSE
USUALLY STRINGS OR NATURAL NUMBERS

决策问题

设S为兴趣子集，可能带有有序元素

对于某些元素x，x是否在S中？

问题：有多少自然数的子集？

例1：S是质数集合，x是一个自然数；x是否在S中（x是质数）？
例2：S是一个无向图（相邻对的集合）；S是否是3-可着色的？
例3：S是一个C语言程序；S是否是语法正确的？
例4：S是一个C语言程序；S是否在所有输入下都终止？
例5：S是一个字符串集合；S的语言是否是常规的、上下文自由的，…？
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 a review of automata and grammars
Alphabets and Strings

• DEFINITION 1. An alphabet $\Sigma$ is a finite, non-empty set of abstract symbols.

• DEFINITION 2. $\Sigma^*$, the set of all strings over the alphabet, $\Sigma$, is given inductively as follows.
  
  – Basis: $\lambda \in \Sigma^*$ (the null string is denoted by $\lambda$, it is the string of length 0, that is $|\lambda| = 0$) [text uses $\varepsilon$ 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 ($\lambda$), 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.
Languages

• DEFINITION 3. Let $\Sigma$ be an alphabet. A *language over* $\Sigma$ is a subset, $L$, of $\Sigma^*$.

• Example. Languages over the alphabet $\Sigma = \{a, b\}$.
  - $\emptyset$ (the empty set) is a language over $\Sigma$
  - $\Sigma^*$ (the universal set) is a language over $\Sigma$
  - $\{a, bb, aba\}$ (a finite subset of $\Sigma^*$) is a language over $\Sigma$.
  - $\{ab^n a^m | n = m^2, n, m \geq 0\}$ (infinite subset) is a language over $\Sigma$.

• DEFINITION 4. Let $L$ and $M$ be two languages over $\Sigma$. Then the *concatenation of $L$ with $M$*, denoted $L \cdot M$ is the set,
  $$L \cdot M = \{ x \cdot y | x \in L \text{ and } y \in M \}$$

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 $x \cdot y = x' \cdot (a \cdot y)$. 

© UCF EECS
UNIVERSE OF LANGUAGES

Non-RE = Semi-Dec = Phrase-Structured

Recursive = Decidable

Context-Sensitive

Context-Free

DCFL

REGULAR
REWRITING SYSTEMS

GRAMMARS

Type 0 = Phrase-Structured

Type 1 = Context-Sensitive

Type 2 = Context-Free

Deterministic CFG

LR(k)

Type 3 = Regular = Right Linear
AUTOMATA
Recognizers that use State & storage
Turing Machines (DTM = NDTM)

LBAs (DLBAs = NDLBAs)

NPDAs

DPDAs

DFAs = NDFAs

Of these models, only TMs can do general computation
What We are Studying

Computability Theory

The study of what can/cannot be done via purely computational means.

Complexity Theory

The study of what can/cannot be done well via purely computational means.
Graph Coloring

• Instance: A graph \( G = (V, E) \) and an integer \( k \).
• Question: Can \( G \) be "properly colored" with at most \( k \) colors?

• Proper Coloring: a color is assigned to each vertex so that adjacent vertices have different colors.

• Suppose we have two instances of this problem (1) is True (Yes) and the other (2) is False (No).

• AND, you know (1) is Yes and (2) is No. (Maybe you have a secret program that has analyzed the two instance.)
Checking a “Yes” Answer

- Without showing how your program works (you may not even know), how can you convince someone else that instance (1) is, in fact, a Yes instance?

- We can assume the output of the program was an actual coloring of $G$. Just give that to a doubter who can easily check that no adjacent vertices are colored the same, and that no more than $k$ colors were used.

- How about the No instance?

- What could the program have given that allows us to quickly "verify" (2) is a No instance?

  - No One Knows!!
Checking a “No” Answer

- The only thing anyone has thought of is to have it test all possible ways to k-color the graph – all of which fail, of course, if “No” is the correct answer.
- There are an exponential number of things (colorings) to check.
- For some problems, there seems to be a big difference between verifying Yes and No instances.
- To solve a problem efficiently, we must be able to solve both Yes and No instances efficiently.
Hard and Easy

• **True Conjecture:** If a problem is easy to solve, then it is easy to verify (just solve it and compare).

• **Contrapositive:** If a problem is hard to verify, then it is (probably) hard to solve.

• There is nothing magical about Yes and No instances – sometimes the Yes instances are hard to verify and No instances are easy to verify.

• And, of course, sometimes both are hard to verify.
Easy Verification

• Are there problems in which both Yes and No instances are easy to verify?

• Yes. For example: Search a list $L$ of $n$ values for a key $x$.
  • Question: Is $x$ in the list $L$?

• Yes and No instances are both easy to verify.

• In fact, the entire problem is easy to solve!!
Verify vs Solve

- Conjecture: If both Yes and No instances are easy to verify, then the problem is easy to solve.

- No one has yet proven this claim, but most researchers believe it to be true.

- Note: It is usually relatively easy to prove something is easy – just write an algorithm for it and prove it is correct and that it is fast (usually, we mean polynomial).

- But, it is usually very difficult to prove something is hard – we may not be clever enough yet. So, you will often see "appears to be hard."
Instances vs Problems

• Each instance has an 'answer'.
  – An instance’s answer is the solution of the instance - it is *not* the solution of the problem.
  – A solution of the problem is a computational procedure that finds the answer of any instance given to it – the procedure must halt on all instances – it must be an 'algorithm'.

2/8/19 © UCF EECS
Three Classes of Problems

Problems can be classified to be in one of three groups (classes):

Undecidable, Exponential, and Polynomial.

Theoretically, all problems belong to exactly one of these three classes and our job is often to find which one.
Why do we Care?

When given a new problem to solve (design an algorithm for), if it's undecidable, or even exponential, you will waste a lot of time trying to write a polynomial solution for it!!

If the problem really is polynomial, it will be worthwhile spending some time and effort to find a polynomial solution and, better yet, the lowest degree polynomial solution.

You should know something about how hard a problem is before you try to solve it.
Procedure (Program)

- A finite set of operations (statements) such that
  - Each statement is finitely presented and formed from a predetermined finite set of symbols and is constrained by some set of language syntax rules.
  - The current state of the machine model is finitely presentable.
  - The semantic rules of the language specify the effects of the operations on the machine’s state and the order in which these operations are executed.
  - If the procedure (eventually) halts when started on some input, it produces the correct answer to this given instance of the problem.
Algorithm

• A procedure that
  – Correctly solves any instance of a given problem.
  – Completes execution in a finite number of steps no matter what input it receives.
Sample Algorithm/Procedure

{ Example algorithm:
  Linear search of a finite list for a key;
  If key is found, answer “Yes”;
  If key is not found, answer “No”; }

{ Example procedure:
  Linear search of a finite list for a key;
  If key is found, answer “Yes”;
  If key is not found, try this strategy again; }

Note: Latter is not unreasonable if the list can be increased in size by some properly synchronized concurrent thread.
Procedure vs Algorithm

Looking back at our approaches to “find a key in a finite list,” we see that the algorithm always halts and always reports the correct answer. In contrast, the procedure does not halt in some cases, but never lies.

What this illustrates is the essential distinction between an algorithm and a procedure – algorithms always halt in some finite number of steps, whereas procedures may run on forever for certain inputs. A particularly silly procedure that never lies is a program that never halts for any input.
Notion of Solvable

• A problem is solvable if there exists an algorithm that solves it (provides the correct answer for each instance).
• The fact that a problem is solvable or, equivalently, decidable does not mean it is solved. To be solved, someone must have actually produced a correct algorithm.
• The distinction between solvable and solved is subtle. Solvable is an innate property – an unsolvable problem can never become solved, but a solvable one may or may not be solved in an individual’s lifetime.
An Old Solvable Problem

Does there exist a set of positive whole numbers, $a$, $b$, $c$ and an $n>2$ such that $a^n + b^n = c^n$?

In 1637, the French mathematician, Pierre de Fermat, claimed that the answer to this question is “No”. This was called Fermat’s Last Theorem, despite the fact that he never produced a proof of its correctness.

While this problem remained unsolved until Fermat’s claim was verified in 1995 by Andrew Wiles, the problem was always solvable, since it had just one question, so the solution was either “Yes” or “No”, and an algorithm exists for each of these candidate solutions.
Research Territory

Decidable – vs – Undecidable
(area of Computability Theory)

Exponential – vs – polynomial
(area of Computational Complexity)

For “easy” problems, we want to determine lower and upper bounds on complexity and develop best Algorithms
(area of Algorithm Design/Analysis)
A CS Grand Challenge

Does $P=NP$?

There are many equivalent ways to describe $P$ and $NP$. For now, we will use the following.

$P$ is the set of decision problems (those whose instances have “Yes”/ “No” answers) that can be solved in polynomial time on a deterministic computer (no concurrency or guesses allowed).

$NP$ is the set of decision problems that can be solved in polynomial time on a non-deterministic computer (equivalently one that can spawn an unbounded number of parallel threads; equivalently one that can be verified in polynomial time on a deterministic computer).

Again, as “Does $P=NP$?” has just one question, it is solvable, we just don’t yet know which solution, “Yes” or “No”, is the correct one.
Computability vs Complexity

Computability focuses on the distinction between solvable and unsolvable problems, providing tools that may be used to identify unsolvable problems – ones that can never be solved by mechanical (computational) means. Surprisingly, unsolvable problems are everywhere as you will see.

In contrast, complexity theory focuses on how hard it is to solve problems that are known to be solvable. Hard solvable problems abound in the real world. We will address computability theory for the first part of this course, returning to complexity theory later in the semester.
REVIEW
REGULAR LANGUAGES
Regular Languages # 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
    • \(\lambda\)-Closure
    • Subset construction
    • Reachable states
    • Reaching states
    • Minimizing DFAs (distinguishable states)
Regular Languages # 2

• Regular expressions and Regular Sets
  – Definition of regular expressions and regular sets
  – Every regular set is a regular language
  – Every regular language is a regular set

• Ripping states (GNFA)

• $R_{i,j}^k$ expressions
  – $R_{i,j}^{k+1} = (R_{ij}^k + R_{ik}^k \cdot (R_{kk}^k)^* \cdot R_{kj})$
  – $L(A) = +_{f \in F} R_{1f}^n$

• Regular equations
  – Uniqueness of solution to $R=Q+RP$
  – Solving for expressions associated with states
Regular Languages # 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 (Adversarial Nature)

• Minimization (using distinguishable states)

• Myhill-Nerode
  – Right Invariant Equivalence Relations (RIER)
  – Specific RIER, \( x R_L y \) \( \forall z \) \( [xz \in L \iff yz \in L] \) is minimal
  – Uniqueness of minimum state DFA based on \( R_L \)
  – Use to show languages are no Regular
Regular Languages # 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
Regular Languages # 5

• 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
REVIEW CONTEXT-FREE & CONTEXT-SENSITIVE LANGUAGES
Context-Free #1

• 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
Context-Free #2

- Context free grammars
  - 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
Context-Free #3

• 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
Context-Free #4

• Closure
  – Union, concatenation, star
  – Substitution
  – Intersection with regular
  – Quotient with regular, Prefix, Suffix, Substring

• Non-Closure
  – Intersection, complement, min, max
Context-Sensitive

• Context sensitive grammars and LBAs
  – Rules for CSG
  – Ability to shuttle symbols to preset places
  – Just basic definition of LBA
Concrete Model of FSA

L is a finite state (regular) language over finite alphabet $\Sigma$
Each $x_i$ is a character in $\Sigma$
w = $x_1 x_2 \ldots x_n$ is a string to be tested for membership in L

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$\ldots$</th>
<th>$x_{n-1}$</th>
<th>$x_n$</th>
</tr>
</thead>
</table>

$q_0$  

- 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 string as member of L if it ends up in a state from Final State set $F \subseteq Q$. 

© UCF EECS
Finite State Automata

• 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 \( A \)
  - \( \Sigma \) is a finite set of symbols called the alphabet of \( A \)
  - \( \delta \) 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 \( \delta \), \( \delta^* : Q \times \Sigma^* \rightarrow Q \), by

  – \( \delta^*(q, \lambda) = q \) where \( \lambda \) is the string of length 0
    • Note that text uses \( \varepsilon \) rather than \( \lambda \) 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 \( \delta \), \( \delta^+ \), by

  – \( \delta^+(q,w) = \delta^*(q,w) \) when \(|w|>0\) or, equivalently, \( w \in \Sigma^+ \)

• The function \( \delta^* \) 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., \( q \)
  – The state \( q_0 \) has an arc entering it with no source, e.g., \( q_0 \)
  – 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., \( q \xrightarrow{a} s \)
  – Each final state has an extra circle around its node, e.g., \( f \)
Sample DFAs # 1, 2

\( \mathcal{A} = ( \{E, O\}, \{0, 1\}, \delta, E, \{O\}) \), where \( \delta \) 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 \( \delta' \) 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

$A'' = (\{0,1,2\}, \{0,1\}, \delta'', 0, \{2\})$, where $\delta''$ is defined by above diagram. $L(A'') = \{w | w$ 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 3, has a remainder of 2 $\}$
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 ($f$)
- 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)
where $\delta'''$ 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, but have put this up so you can see the pattern.
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.
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 $A_3 = (Q_1 \times Q_2, \Sigma, \delta_3, <q, s>, F_3)$ where

  $\delta_3(<q, s>, a)= <\delta_1(q, a), \delta_2(s, a)> \text{, } q \in Q_1, s \in Q_2, a \in \Sigma$

  - $L(A_1) \cup L(A_2)$ 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 $A^C = (Q, \Sigma, \delta, q_0, Q-F)$
- $L(A^C) = \{ w | \delta^*(q_0, w) \in Q-F \} = \{ w | \delta^*(q_0, w) \notin F \} = \{ w | w \notin L(A) \}$
- Again, 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 $Q_1 \cap Q_2 = \emptyset$
- $B = (Q_1 \times Q_2, \Sigma, \delta_3, <q_0, s_0>, F_3)$ where $\delta_3(<q, s>, a) = <\delta_1(q, a), \delta_2(s, a)>$
- 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
  - $\Sigma$ is a finite set of symbols called the alphabet of A
  - $\delta$ is a function from $Q \times \Sigma_e$ into $P(Q) = 2^Q$; Note: $\Sigma_e = (\Sigma \cup \{\lambda\})$
    - $\delta: Q \times \Sigma_e \to P(Q))$ 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
    - $\delta(S, a) = \bigcup_{q \in S} \delta(q, a)$, where $a \in \Sigma_e$ — if $S = \emptyset$, $\bigcup_{q \in S} \delta(q, a) = \emptyset$
NFA Transitions

• Given an NFA, \( A = (Q, \Sigma, \delta, q_0, F) \), we can define the reflexive transitive closure of \( \delta \), \( \delta^*: \mathcal{P}(Q) \times \Sigma^* \rightarrow \mathcal{P}(Q) \), by
  - \( \lambda\)-Closure(\( S \)) = \{ \( t \mid t \in \delta^*(S, \lambda) \} \), \( S \in \mathcal{P}(Q) \) – extended \( \delta \)
  - \( \delta^*(S, \lambda) = \lambda\)-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 \( \delta \), \( \delta^* \), by
  - \( \delta^*(S, w) = \delta^*(S, w) \) when \( |w| > 0 \) or, equivalently, \( w \in \Sigma^* \)

• The function \( \delta^* \) 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 $\lambda$
  – 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 $\text{seq}(S)=<q_1,q_2,q_3>$. If $S=\emptyset$ then $\text{seq}(S)=<>$
- 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 $\lambda$-Closure$(q) = \{ t \mid t \in \delta^*(q, \lambda) \}$
- We can extend this to $S \in P(Q)$ by
  $\lambda$-Closure$(S) = \{ t \mid t \in \delta^*(q, \lambda), q \in S \} = \{ t \mid t \in \lambda$-Closure$(q), q \in S \}$

![Diagram of λ-Closure](image)

<table>
<thead>
<tr>
<th>State</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>λ-closure</td>
<td>{A}</td>
<td>{B, C}</td>
<td>{C}</td>
<td>{D, E}</td>
<td>{E}</td>
</tr>
</tbody>
</table>
Details of DFA

- Let $A = (Q, \Sigma, \delta, q_0, F)$ be an arbitrary NFA
- In an abstract sense, $A' = (<P(Q)>, \Sigma, \delta', <\lambda\text{-Closure}({q_0})>, F')$, 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 $\delta'(S,a) = <\lambda\text{-Closure}(\delta(S,a))> = \cup_{q \in S} \lambda\text{-Closure}(\delta(q,a))$, where $a \in \Sigma$, $S \in P(Q)$
- $F' = \{ S \in <P(Q)> \mid (S \cap F) \neq \emptyset \}$
Regular Languages and NFAs

• Showing that 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
Convert from NFA to DFA
Lexical Analysis

• Consider distinguishing variable names from keywords like IF, THEN, ELSE, etc.
• This really screams for non-determinism
• Non deterministic automata typically have fewer states
• However, non-deterministic FSA interpretation is not as fast as deterministic
Practice Problems

Practice

1. Using DFA’s (not any equivalent notation) show that the Regular Languages are closed under Min, where Min(L) = \{ w | w \in L, but no proper prefix of w is in L\}.. This means that w \in Min(L) iff w \in L and for no y\neq\lambda is x in L, where w=xy. Said a third way, w is not an extension of any element in L.

2. a.) Present a transition diagram for an NFA for the language associated with the regular expression \((1011 + 111 + 101)^*\).

b.) Use the standard conversion technique (subsets of states) to convert the NFA from (a) to an equivalent DFA. Be sure to not include unreachable states.
Practice DFA/NFA

1. Present a transition diagram for a DFA that recognizes the set of binary strings that, when interpreted as entering the DFA most to least significant digit, each represents a binary number that is divisible by either 2 or 3 or both. Thus, 100, 110, 1001 and 1100 are in the language, but 01, 101, 111 and 1011 are not.

2. a.) Present a transition diagram with no lambda transitions for an NFA associated with the regular expression \((0111 + 111 + 011)^*\). Your NFA must have no more than four states.
   b.) Use the standard conversion technique (subsets of states) to convert the NFA from (a) to an equivalent DFA. Be sure to not include unreachable states.
Regular Expressions

• Primitive:
  - $\Phi$ denotes $\{\}$
  - $\lambda$ denotes $\{\lambda\}$
  - $a$ where $a$ is in $\Sigma$ 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 $RS = \{ xy | x \text{ is in } R \text{ and } y \text{ is in } S \}$
    - $R + S$ denotes $R \cup S = \{ x | x \text{ is in } R \text{ or } x \text{ is in } S \}$
    - $R^*$ denotes $R^*$

• Parentheses are used as needed
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:
  – \( \Phi \) denotes \{\}\n  – \( \lambda \) denotes \{\lambda\}
  – \( a \) where \( a \) is in \( \Sigma \) denotes \{a\}

• Closure: (Assume that R’s and S’s states do not overlap)
  – \( R \cdot S \) start with machine for R, add \( \lambda \) 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 \( \lambda \) 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 \( \lambda \) 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 $Q = \{q_1, q_2, \ldots, q_n\}$
- $R_{ij}^k = \{w | \delta^*(q_i, w) = q_j, \text{ and no intermediate state visited between } q_i \text{ and } q_j, \text{ while reading } w, \text{ has index } > k\}$
- Basis: $k=0$, $R_{ij}^0 = \{ a | \delta(q_i, a) = q_j \}$ sets are either $\Phi$, $\lambda$, or an element of $\Sigma$ 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: $k+1$, $R_{ij}^{k+1} = (R_{ij}^k + R_{ik+1}^k \cdot (R_{k+1k+1}^k)^* \cdot R_{k+1j}^k)$
- $L(A) = \bigoplus_{f \in F} R_{1f}^n$
Convert to RE
\[
\begin{align*}
\cdot R_{11}^0 &= \lambda & R_{12}^0 &= 0 & R_{13}^0 &= \phi \\
\cdot R_{21}^0 &= 0 & R_{22}^0 &= \lambda + 1 & R_{23}^0 &= 0 + 1 \\
\cdot R_{31}^0 &= \phi & R_{32}^0 &= 1 & R_{33}^0 &= \lambda + 1 \\
\cdot R_{11}^1 &= \lambda & R_{12}^1 &= 0 & R_{13}^1 &= \phi \\
\cdot R_{21}^1 &= 0 & R_{22}^1 &= \lambda + 1 + 00 & R_{23}^1 &= 0 + 1 \\
\cdot R_{31}^1 &= \phi & R_{32}^1 &= 1 & R_{33}^1 &= \lambda + 1 \\
\cdot R_{11}^2 &= \lambda + 0(1+00)^*0 & R_{12}^2 &= 0(1+00)^* & R_{13}^2 &= 0(1+00)^*(0+1) \\
\cdot R_{21}^2 &= (1+00)^*0 & R_{22}^2 &= (1+00)^* & R_{23}^2 &= (1+00)^*(0+1) \\
\cdot R_{31}^2 &= 1(1+00)^*0 & R_{32}^2 &= 1(1+00)^* & R_{33}^2 &= \lambda+1+1(1+00)^*(0+1) \\
\cdot L &= R_{12}^3 = \\
&= 0(1+00)^* + 0(1+00)^*(0+1) (1+1(1+00)^*(0+1)) * 1(1+00)^* 
\end{align*}
\]
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 $\lambda$–transition to existing start state.
- Add a new final state $q_f$ and insert $\lambda$–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 is 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 e1 be the regular expression on the arc from some node A to some node B (A≠B); e2 be the expression from B back to B (or λ if there is no recursive arc); e3 be the expression on the arc from B to some other node C (C ≠B but C could be A); e4 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 e4 + e1 e2* e3
• 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

Graph:

- States: q0, q1, q2, qf
- Transitions:
  - q0 to q1: λ
  - q1 to q2: 0
  - q2 to qf: λ
  - q0 to q2: 0
  - q2 to qf: λ
  - q1 to qf: 1+(0+1)1^+
Use Ripping; Rip q2

\[ L = 0 \ (1+(0+1)1^+00)^* = 0 \ \\
(1+(0+1)1^+00)^* \]
Regular Equations

• 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
• \( R = QP^* \)
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, $R = Q + RP = Q + (Q + RP)P$
$= Q + QP + RP^2 = Q + QP + (Q + RP)P^2$
$= Q + QP + QP^2 + RP^3$
$\ldots$
$= Q(\lambda + P + P^2 + \ldots + P_i) + RP^{i+1}$, for all $i \geq 0$
Choose any $w$ in $R$, where $|W| = k$. Then, from above,
$R = Q(\lambda + P + P^2 + \ldots + P^k) + RP^{k+1}$
but, since $P$ does not contain the string of length zero, $w$ is not in $RP^{k+1}$. But then $w$ is in
$Q(\lambda + P + P^2 + \ldots + P^k)$ 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, $Q = \lambda + B_1; P = 0$
  
  $A = QP^* = (\lambda + B_1)0^*$
  
  $= B_10^* + 0^*$
- $B = B_10^*1 + B_0 + 0^*1$

  For $B$, $Q = 0^*1; P = B_10^*1 + B_0 = B(10^*1 + 0)$
- 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)^*

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.
State Minimization

• Text makes it an assignment on Page 299 in Sipser 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 a 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 <1,3>, <2,4,5>, <6>; <1,3> is start and not accept; others are accept
- Write new diagram
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 $\lambda$-transitions to multiple starts.
Substitution

• A substitution is a function, $f$, from each member, $a$, of an alphabet, $\Sigma$, 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 $\Sigma$ 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
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

\[
L/R = \{ x | \exists y \in R \text{ where } xy \in L \}
\]

- Define \( \Sigma' = \{ a' | a \in \Sigma \} \)
- 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 $\text{INIT}(L) = L / \Sigma^*$

- $\text{LAST}(L) = \{ y \mid \exists x \in \Sigma^* \text{ where } xy \in L \}$
  - $\text{LAST}(L) = h( f(L) \cap ( g(\Sigma^*) \cdot \Sigma^* ) )$

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

- $\text{EXTERIOR}(L) = \{ xz \mid \exists y \in \Sigma^* \text{ where } xyz \in L \}$
  - $\text{EXTERIOR}(L) = h( f(L) \cap ( \Sigma^* \cdot g(\Sigma^*) \cdot \Sigma^* ) )$
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**

- **Reachable from** \( (q) = \{ p \mid \exists w \ni \delta(q,w)=p \} \)
  - Just do depth first search from \( q \), marking all reachable states. Works for NFA as well.

- **Reaching to** \( (q) = \{ p \mid \exists w \ni \delta(p,w)=q \} \)
  - Do depth first from \( q \), going backwards on transitions, marking all reaching states. Works for NFA as well.
Min and Max

• Min(L) = \{ w | w ∈ L and no proper prefix of w is in L \} = \\
  \{ w | w ∈ L and if w = xy, x ∈ Σ^*, y ∈ Σ^+ then x ∉ L \}

• Max(L) = \{ w | w ∈ L and w is not the proper prefix of any word in L \} = \\
  \{ w | w ∈ L and if y ∈ Σ^+ then wy ∉ L \}

• Examples:
  – Min(0(0+1)*) = \{0\}
  – Max(0(0+1)*) = \{\}
  – Min(01 + 0 + 10) = \{0,10\}
  – Max(01 + 0 + 10) = \{01,10\}
  – Min(\{a^i b^j c^k | i ≤ k or j ≤ k\}) = \{a^i b^j c^k | | i,j ≥ 0, k = \text{min}(i, j)\}
  – Max(\{a^i b^j c^k | i ≤ k or j ≤ k\}) = \{\} because k has no bound
  – Min(\{a^i b^j c^k | i ≥ k or j ≥ k\}) = \{λ\}
  – Max(\{a^i b^j c^k | i ≥ k or j ≥ k\}) = \{a^i b^j c^k | | i,j ≥ 0, k = \text{max}(i, j)\}
Regular Closed under Min

• Assume L is regular then Min(L) is regular
• Let L = L(A), where A = (Q, Σ, δ, q₀, F) is a DFA with no state unreachable from q₀
• Define A_{min} = (Q∪{dead}, Σ, δ_{min}, q₀, F), where for a ∈ Σ
  \[ δ_{min}(q,a) = δ(q,a), \text{ if } q ∈ Q-F; \ δ_{min}(q,a) = \text{dead, if } q ∈ F; \]
  \[ δ_{min}(\text{dead},a) = \text{dead} \]

The reasoning is that the machine \( A_{\text{min}} \) accepts only elements in L that are not extensions of shorter strings in L. By making it so transitions from all final states in \( A_{\text{min}} \) 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 Max(L) is regular
- Let L = L(A), where A = (Q, Σ, δ, q₀, F) is a DFA with no state unreachable from q₀
- Define A_max = (Q, Σ, δ, q₀, F_max), where
  
  \[ F_{\text{max}} = \{ f \mid f \in F \text{ and } \text{Reachable}^{+}(f) \cap F = \emptyset \} \]

  where Reachable^{+}(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 Reachable^{+} is:

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

Therefore, Regular Languages are closed under Max.
Practice $R_{ij}^k$

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.
Practice Minimization

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

Construct and write down your new, equivalent automaton!!
Pumping Lemma Concept

• Let $A = (Q, \Sigma, \delta, q_1, F)$ be a DFA, where $Q = \{q_1, q_2, \ldots, q_N\}$
• 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 $i \geq 0$, $x y^i z \in L$

- 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 $w = v_1\ldots v_j\ldots v_k\ldots v_m$, where $m = |w|$, and $\delta(q_0,v_1\ldots v_j)=\delta(q_0,v_1\ldots v_k)$, $k > j$, and let this state represent the first one repeated while $A$ consumes $w$.

• Define $x = v_1\ldots v_j$, $y = v_{i+1}\ldots v_k$, and $z = v_{k+1}\ldots v_m$. 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 $\delta(q_0,xy^iz) \in F$, for all $i \geq 0$.

• Consequently, if $w \in L$, $|w|\geq N$, then $w$ can be written in the form $xyz$, where $|xy| \leq N$, $|y| > 0$, and for all $i \geq 0$, $xy^iz \in L$.
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)

- $L = \{ xw x \mid x,w \in \{a,b\}^+ \}$:
  - 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 $\forall \ i \geq 0 \ xy^i z \in L$
  - YOU: Choose $i = 2$
  - PL: $xy^2z = xyyz \in L$
  - 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 = $\{a^{\text{Fib}(k)} | k>0\}$:**
- 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 $|s| \geq N$ (Fib(N+3)>N in all cases; actually 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 Fib(N+3) and Fib(N+3)+N, but the smallest Fibonacci greater than Fib(N+3) is Fib(N+3)+Fib(N+2) and 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 Fib(2) and Fib(3) are within N (1) of 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 \}
  – \{ w w^R \mid w \in \{a,b\}^+ \}
  – \{ 1^{n^2} \mid n > 0 \}
  – \{ w w \mid w \in \{a,b\}^+ \}
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 $\forall z \ [ x R y$ implies $xz R yz ]$.

Note: This is only meaningful for relations over strings.
Myhill-Nerode 1 ⇒ 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 $L$ 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 $x$. 

2/8/19
2. Assume $L$ is the union of some of the classes of a right invariant equivalence relation, $R$, of finite index.

3. Since $x \sim_\leftarrow y$ iff $\forall z \left[ xz \sim_\rightarrow yz \right]$, $R$ is right invariant and $L$ is the union of some of the equivalence classes, then $x \sim_\leftarrow y \Rightarrow \forall z \left[ xz \in L \iff yz \in L \right] \Rightarrow x \sim_{RL} y$.

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 ⇒ 1

3. Assume the specific right invariance equivalence relation $R_L$ where $x R_L y$ iff $\forall z \ [ xz \in L \iff yz \in L ]$ has finite index

1. Define the automaton $A = (Q, \Sigma, \delta, q_1, F)$ by
   $Q = \{ [x]_{RL} \mid x \in \Sigma^* \}$
   $\delta([x]_{RL}, a) = [xa]_{RL}$
   $q_1 = [\lambda]$
   $F = \{ [x]_{RL} \mid x \in L \}$

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 $x \ R_L y \iff \forall z \ [ xz \in L \iff yz \in L ]$.
- 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 $a^j b b^{j-1} \notin L$ 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 $[a^ib]$, $i > 0$.
- It's clear that $a^iba^ib$ is in the language, but $a^kbaa^ib$ is not when $k < i$.
- This shows that there is a separate equivalence class, $[a^ib]$, 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
  \[ x R_L y \iff \forall z \ [ xz \in L \iff yz \in L ] \]

• 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
  – \( M = (Q, \Sigma, \Gamma, \delta, \gamma, q_0) \)
    \( \Gamma \) 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
  – \( M = (Q, \Sigma, \Gamma, \delta, \gamma, q_0) \)
    \( \Gamma \) 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.
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. $L = \{ x#y \mid x, y \in \{0,1\}^+ \text{ and } y \text{ is the ones complement of } x \}$

   b. $L = \{ a^i b^j c^k \mid i > j \times k \}$

   c. $L = \{ x w x \mid x, w \in \{a,b\}^+ \text{ Here } |x|>0 \text{ and } |w|>0 \}$

2. Write a regular (right linear) grammar that generates $L = \{ w \mid w \in \{0,1\}^+ \text{ and } w \text{ interpreted as a binary number is divisible by either 2 or 3 or both.} \}$

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 1001 to $x$ (assumes all numbers are positive, including results). Assume that $x$ is read starting with its least significant digit.

Examples: 00010 → 01011; 00101 → 01110; 00111 → 10000; 00110 → 01111
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 from 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
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).
Formalism for Grammars

Definition: A language is a set of strings of characters from some alphabet. The strings of the language are called sentences or statements.

A string over some alphabet is a finite sequence of symbols drawn from that alphabet.

A meta-language is a language that is used to describe another language.

A very well known meta-language is BNF (Backus Naur Form)

It was developed by John Backus and Peter Naur, in the late 50s, to describe programming languages.

Noam Chomsky in the early 50s developed context free grammars that can be expressed using BNF.
Grammars

- $G = (V, \Sigma, R, S)$ is a Phrase Structured Grammar (PSG) where
  - $V$: Finite set of non-terminal symbols
  - $\Sigma$: Finite set of terminal symbols
  - $R$: finite set of rules of form $\alpha \rightarrow \beta$,
    - $\alpha$ in $(V \cup \Sigma)^* V (V \cup \Sigma)^*$
    - $\beta$ in $(V \cup \Sigma)^*$
  - $S$: a member of $V$ called the start symbol
- Right linear restricts all rules to be of forms
  - $\alpha$ in $V$
  - $\beta$ of form $\Sigma V$, $\Sigma$ or $\lambda$
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:
  \[ A \rightarrow a, \quad A \in V, \ a \in \Sigma^* \]
  \[ A \rightarrow \lambda, \quad A \in V, \ a \in \Sigma^* \]
  \[ A \rightarrow aB, \quad A, \ B \in V, \ a \in \Sigma^* \]
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 $q \to \text{as}$ iff $\delta(q,a) = s$
  $q \to \lambda$ iff $q \in F$
Example of DFA to Grammar

- **DFA**

- **Grammar**

  \[
  \begin{align*}
  A &\rightarrow 0 \, B \mid 1 \, B \\
  B &\rightarrow 0 \, A \mid 1 \, C \mid \lambda \\
  C &\rightarrow 0 \, C \mid 1 \, A \mid \lambda
  \end{align*}
  \]
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 $\delta$ is defined by
  - $\delta(A, a) \subseteq \{B\}$ if $A \rightarrow aB$
  - $\delta(A, a) \subseteq \{f\}$ if $A \rightarrow a$
  - $\delta(A, \lambda) \subseteq \{f\}$ if $A \rightarrow \lambda$
Example of Grammar to NFA

• Grammar

\[
\begin{align*}
S & \rightarrow 0\ S \mid 1\ A \\
A & \rightarrow 0\ S \mid 0\ A \mid 1\ B \mid \lambda \\
B & \rightarrow 1\ S \mid 0\ B
\end{align*}
\]

• DFA

![Diagram of NFA with states S, A, B and transitions labeled with 0s and 1s]
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:
  – $S \rightarrow aT$
  – $T \rightarrow Sb | b$
• $L(G) = \{ a^n b^n | n > 0 \}$ which is a classic non-regular, context-free language
Context Free Languages
Context Free Grammar

G = (V, Σ, R, S) is a PSG where
Each member of R is of the form
A → α where α is a strings (V∪Σ)*
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.
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).
A sentence generation is called a derivation.

Grammar for a simple assignment statement:

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

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

The statement \(a := b \ast (a + c)\) is generated by the leftmost derivation:

\(<\text{assgn}>\) \(\Rightarrow\) \(<\text{id}>\) := \(<\text{expr}>\)  
\(\Rightarrow\) \(a := \ <\text{expr}>\)  
\(\Rightarrow\) \(a := <\text{id}> \ast <\text{expr}>\)  
\(\Rightarrow\) \(a := b \ast <\text{expr}>\)  
\(\Rightarrow\) \(a := b \ast (\ <\text{expr}>\ )\)  
\(\Rightarrow\) \(a := b \ast (\ <\text{id}> + <\text{expr}>\ )\)  
\(\Rightarrow\) \(a := b \ast (\ a + <\text{expr}>\ )\)  
\(\Rightarrow\) \(a := b \ast (\ a + <\text{id}>\ )\)  
\(\Rightarrow\) \(a := b \ast (\ a + c )\)
A parse tree is a graphical representation of a derivation. For instance, the parse tree for the statement \( a := b \ast (a + c) \) is:

```
<assign>
  <id>
  a
</id>
  :=
</assign>

<expr>
  <id>
  b
</id>
  *
</expr>

<expr>
  +
  <id>
  a
</id>
  <expr>
  c
</expr>
</expr>
```

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 \( a := b + c \ast a \).

\[
\begin{align*}
\text{<assign>} & \rightarrow \text{id} := \text{<expr>} \\
\text{id} & \rightarrow a \mid b \mid c \\
\text{<expr>} & \rightarrow \text{<expr>} + \text{<expr>} \\
& \mid \text{<expr>} \ast \text{<expr>} \\
& \mid (\text{<expr>} ) \\
& \mid \text{id}
\end{align*}
\]
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.
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:

\[
\begin{align*}
<\text{assign}> & \rightarrow <\text{id}> := <\text{expr}> \\
<\text{id}> & \rightarrow a \mid b \mid c \\
<\text{expr}> & \rightarrow <\text{expr}> + <\text{term}> \\
& \quad \mid <\text{term}> \\
<\text{term}> & \rightarrow <\text{term}> \ast <\text{factor}> \\
& \quad \mid <\text{factor}> \\
<\text{factor}> & \rightarrow ( <\text{expr}> ) \\
& \quad \mid <\text{id}>
\end{align*}
\]

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.
Leftmost derivation:
<assign> → <id> := <expr>
  → a := <expr>
  → a := <expr> + <term>
  → a := <term> + <term>
  → a := <factor> + <term>
  → a := <factor> + <term>
  → a := <id> + <term>
  → a := b + <term>
  → a := b + <term> * <factor>
  → a := b + <factor> * <factor>
  → a := b + <id> * <factor>
  → a := b + c * <factor>
  → a := b + c * <id>
  → a := b + c * a

Rightmost derivation:
<assign> ⇒ <id> := <expr>
  ⇒ <id> := <expr> + <term>
  ⇒ <id> := <expr> + <term> * <factor>
  ⇒ <id> := <expr> + <term> * <id>
  ⇒ <id> := <expr> + <term>* a
  ⇒ <id> := <expr> + <factor>* a
  ⇒ <id> := <expr> + <id> * a
  ⇒ <id> := <expr> + c * a
  ⇒ <id> := <term> + c * a
  ⇒ <id> := <factor> + c * a
  ⇒ <id> := <id> + c * a
  ⇒ <id> := b + c * a
  ⇒ 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:

\[
<\text{expr}> \rightarrow <\text{expr}> <\text{op}> <\text{expr}> \mid \text{id} \mid \text{int} \mid (<\text{expr}>)
\]
\[
<\text{op}> \rightarrow + \mid - \mid * \mid / 
\]

Can be rewritten as:

\[
<\text{expr}> \rightarrow <\text{term}> \mid <\text{expr}> + <\text{term}> \mid <\text{expr}> - <\text{term}>
\]
\[
<\text{term}> \rightarrow <\text{factor}> \mid <\text{term}> * <\text{factor}> \mid <\text{term}> / <\text{factor}>
\]
\[
<\text{factor}> \rightarrow \text{id} \mid \text{int} \mid (<\text{expr}>)
\]
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
Removing Left Recursion if doing Top Down

Given left recursive and non left recursive rules

\[ A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m \]

Can view as

\[ A \rightarrow (\beta_1 \mid \ldots \mid \beta_m) (\alpha_1 \mid \ldots \mid \alpha_n)^* \]

Star notation is an extension to normal notation with obvious meaning

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

\[ A \rightarrow \beta_1 B \mid \ldots \mid \beta_m B \]

\[ B \rightarrow \alpha_1 B \mid \ldots \mid \alpha_n B \mid \lambda \]
Right Recursive Expressions

Grammar:  \[ \text{Expr} \rightarrow \text{Expr} + \text{Term} \mid \text{Term} \]
          \[ \text{Term} \rightarrow \text{Term} \ast \text{Factor} \mid \text{Factor} \]
          \[ \text{Factor} \rightarrow (\text{Expr}) \mid \text{Int} \]

Fix:  \[ \text{Expr} \rightarrow \text{Term} \text{ExprRest} \]
      \[ \text{ExprRest} \rightarrow + \text{Term} \text{ExprRest} \mid \lambda \]
      \[ \text{Term} \rightarrow \text{Factor} \text{TermRest} \]
      \[ \text{TermRest} \rightarrow \ast \text{Factor} \text{TermRest} \mid \lambda \]
      \[ \text{Factor} \rightarrow (\text{Expr}) \mid \text{Int} \]
Bottom Up vs Top Down

• Bottom-Up: Two stack operations
  – Shift (move input symbol to stack)
  – Reduce (replace top of stack $\alpha$ 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 $\alpha$, 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:
  \[ A \to a, \quad A \in V, \ a \in \Sigma \]
  \[ A \to BC, \quad A,B,C \in V \]

• If the language contains \( \lambda \) then we allow
  \[ S \to \lambda \]
  and constrain all non-terminating rules of form to be
  \[ A \to BC, \quad A \in V, \ B,C \in V-\{S\} \]
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 $\lambda$-Rules

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

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

Let $G = (V, \Sigma, R, S)$ be an arbitrary CFG that has had its $\lambda$-rules removed, except perhaps from start symbol

Compute 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 $\lambda$-rules and unit-rules removed
• Non-productive non-terminal symbols never lead to a terminal string (not productive)
• $\text{Productive}(G)$ is computed by
  $\text{Productive}(G) \supseteq \{ A \mid A \rightarrow \alpha, \alpha \in \Sigma^* \}$
  Repeat
    $\text{Productive}(G) \supseteq \{ B \mid B \rightarrow \alpha, \alpha \in (\Sigma \cup \text{Productive})^* \}$
  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 $\lambda$-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

\[
\text{Useful}(G) \supseteq \{ S \}
\]

Repeat

\[
\text{Useful}(G) \supseteq \{ C \mid B \rightarrow \alpha C \beta, C \in V \cup \Sigma, B \in \text{Useful}(G) \}
\]

to 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 $\lambda$-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 $G' = (V \cup \{<a>|a \in \Sigma\}, \Sigma, R, S)$
- Add the rules $<a> \rightarrow a$, for all $a \in \Sigma$
- For any rule, $A \rightarrow \alpha, |\alpha| > 1$, change each terminal symbol, $a$, in $\alpha$ to the non-terminal $<a>$
- Now, for each rule $A \rightarrow BC \alpha, |\alpha| > 0$, introduce the new non-terminal $B<C\alpha>$, and replace the rule $A \rightarrow BC \alpha$ with the rule $A \rightarrow B<C\alpha>$ and add the rule $<C\alpha> \rightarrow C\alpha$
- Iteratively apply the above step until all rules are in CNF
Example of CNF Conversion
Starting Grammars

- \( L = \{ a^i b^j c^k \mid i=j \text{ or } j=k \} \)
- \( G = (\{S,A,<B=C>,C,<A=B>\}, \{a,b\}, R, S) \)
- \( R: \)
  - \( S \rightarrow A \mid C \)
  - \( A \rightarrow a A \mid <B=C> \)
  - \( <B=C> \rightarrow b <B=C> c \mid \lambda \)
  - \( C \rightarrow C c \mid <A=B> \)
  - \( <A=B> \rightarrow a <A=B> b \mid \lambda \)
Remove Null Rules

- Nullable = \{<B=C>, <A=B>, A, C, S\}
  - S' → S | λ \\
  - S → A | C \\
  - A → a A | a |<B=C> \\
  - <B=C> → b <B=C> c | b c \\
  - C → C c | c | <A=B> \\
  - <A=B> → a <A=B> b | ab \\
  - S' added to V
Remove Unit Rules

- **Chains=**

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

- \[ S' \rightarrow \lambda | aA | a | b<B=C>c | bc | Cc | c | a<A=B>b | ab \]
- \[ S \rightarrow aA | a | b<B=C>c | bc | Cc | c | a<A=B>b | ab \]
- \[ A \rightarrow aA | a | b<B=C>c | bc \]
- \[ <B=C> \rightarrow b<B=C>c | bc \]
- \[ C \rightarrow Cc | c | a<A=B>b | ab \]
- \[ <A=B> \rightarrow a<A=B>b | ab \]
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

- \( S' \rightarrow \lambda | \langle a \rangle A | a | \langle b \rangle \langle \langle B=C \rangle \langle c \rangle \rangle | \langle b \rangle \langle c \rangle \ | C \langle c \rangle | c | \langle a \rangle \langle \langle A=B \rangle \langle b \rangle \rangle | \langle a \rangle \langle b \rangle \)
- \( A \rightarrow \langle a \rangle A | a | \langle b \rangle \langle \langle B=C \rangle \langle c \rangle \rangle | \langle b \rangle \langle c \rangle \)
- \( \langle B=C \rangle \rightarrow \langle b \rangle \langle \langle B=C \rangle \langle c \rangle \rangle | \langle b \rangle \langle c \rangle \)
- \( C \rightarrow C \langle c \rangle | c | \langle a \rangle \langle \langle A=B \rangle \langle b \rangle \rangle | \langle a \rangle \langle b \rangle \)
- \( \langle A=B \rangle \rightarrow \langle a \rangle \langle \langle A=B \rangle \langle b \rangle \rangle | \langle a \rangle \langle b \rangle \)
- \( \langle \langle B=C \rangle \langle c \rangle \rangle \rightarrow \langle B=C \rangle \langle c \rangle \)
- \( \langle \langle A=B \rangle \langle b \rangle \rangle \rightarrow \langle A=B \rangle \langle b \rangle \)
- \( \langle a \rangle \rightarrow a \)
- \( \langle b \rangle \rightarrow b \)
- \( \langle c \rangle \rightarrow c \)
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 \( a_1 \ldots a_n \).
Let the grammar contain \( r \) terminal and nonterminal symbols \( R_1 \ldots R_r \),
Let \( R_1 \) be the start symbol.
Let \( P[n,n,r] \) be an array of Booleans. Initialize all elements of \( P \) to false.
For each \( i = 1 \) to \( n \)
  For each unit production \( R_j \rightarrow a_i \), set \( P[i,1,j] = \text{true} \).
  For each \( i = 2 \) to \( n \)
    For each \( j = 1 \) to \( n-i+1 \)
      For each \( k = 1 \) to \( i-1 \)
        For each production \( R_A \rightarrow R_B R_C \)
          If \( P[j,k,B] \) and \( P[j+k,i-k,C] \) then set \( P[j,i,A] = \text{true} \)
If \( P[1,n,1] \) is true then \( a_1 \ldots a_n \) is member of language
else \( a_1 \ldots a_n \) is not member of language
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 $R$:

<table>
<thead>
<tr>
<th></th>
<th>S → AB</th>
<th>BA</th>
</tr>
</thead>
<tbody>
<tr>
<td>A →</td>
<td>CD</td>
<td>a</td>
</tr>
<tr>
<td>B →</td>
<td>CE</td>
<td>b</td>
</tr>
<tr>
<td>C →</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>D →</td>
<td>AC</td>
<td></td>
</tr>
<tr>
<td>E →</td>
<td>BC</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>b</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A,C</td>
<td>B,C</td>
<td>B,C</td>
<td>A,C</td>
</tr>
<tr>
<td>2</td>
<td>S,D</td>
<td>E</td>
<td>S,E</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>B</td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>S,E</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2\textsuperscript{nd} CKY Example

\[ E \rightarrow \ EF \mid ME \mid PE \mid a \]
\[ F \rightarrow \ MF \mid PF \mid ME \mid PE \]
\[ P \rightarrow \ + \]
\[ M \rightarrow \ - \]

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>–</th>
<th>a</th>
<th>+</th>
<th>a</th>
<th>–</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>E</td>
<td>M</td>
<td>E</td>
<td>P</td>
<td>E</td>
<td>M</td>
<td>E</td>
</tr>
<tr>
<td>2</td>
<td>E, F</td>
<td>E, F</td>
<td>E, F</td>
<td>E, F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>E, F</td>
<td>E, F</td>
<td>E, F</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>E, F</td>
<td>E, F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>E</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( \text{© UCF EECS} \)

2/8/19
Practice CFGs

1. Write a CFG for the following languages:
   \[ L = \{ a^n \ b^m \ c^t | n < m \ or \ m > t \ or \ n = t \} \]

2. Convert the following grammar to a CNF equivalent grammar. Show all steps.
   \[ G = (\{ S, S_1, S_2, B, C\}, \{ a, b, c \}, R, S) \], where \( R \) is:
   \[
   \begin{align*}
   S & \rightarrow S_1 | S_2 \\
   S_1 & \rightarrow a \ S_1 \ b | S_1 \ b | b \\
   S_2 & \rightarrow c \ C \ a \ B \\
   C & \rightarrow c \ C \ a | C \ a | a \\
   B & \rightarrow a \ B \ b | \lambda
   \end{align*}
   \]

3. Present the CKY recognition matrix for the string \( b \ a \ a \ b \ a \) assuming the Chomsky Normal Form grammar \( G = (\{ S, T, B \}, \{ a, b \}, R, S) \), where \( R \) is specified by the rules
   \[
   \begin{align*}
   S & \rightarrow S \ T | T \ S | a \\
   T & \rightarrow B \ S | b \\
   B & \rightarrow B \ T | SS | b
   \end{align*}
   \]
CFL Pumping Lemma Concept

- Let L be a context free language the there is CNF grammar G = (V, Σ, 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 → 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 → a.
- Any terminal branch in a derivation tree of height > |V| has more than |V| internal nodes labelled with non-terminals. The “pigeon hole 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 $|vwy| \leq N$, $|vx|>0$, and for all $i \geq 0$, $uv^ixw^iy \in L$.
- 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, Σ, R, S). Let |V| = k. For any string z, such that |z| ≥ 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 Pigeon Hole 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 ⇒* uTy ⇒* uvTxy ⇒* uvwxy

• Clearly, then, we know S ⇒* uTy; T ⇒* vTx; and T ⇒* w

• But then, we can start with S ⇒* uTy; repeat T ⇒* vTx zero or more times; and then apply T ⇒* w.

• But then, S ⇒* uv^iwx^iy for all i≥0, and thus uv^iwx^iy ∈ L, for all i ≥0.
Visual Support of Proof

\[ S \]

\[ u \quad v \quad w \quad x \quad y \]

\[ T \]

\[ i = 2 \]

\[ i = 0 \]

\[ w \]

\[ S \]

\[ S \]

\[ S \]

\[ u \quad v \quad w \quad x \quad y \]

\[ T \]

\[ T \]

\[ T \]

\[ u \quad v \quad w \quad x \quad y \]

\[ v \quad w \quad x \]
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 $i \geq 0$, $uv^iwx^iy \in L$  
  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. $vwx$ 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. $vwx$ 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
Practice CFL Pumping Lemma

1. Write a CFG to show the language is a CFL or use the Pumping Lemma for CFLs to prove that it is not for each of the following.
   a) \( L = \{ a^i b^j \mid j > i^3, i>0 \} \)
   b) \( L = \{ a^i b^j \mid j < 3^i, i>0 \} \)

2. Consider the context-free grammar \( G = \{ \{ S \}, \{ a, b \}, R, S \} \)
   \( R: \)
   \( S \rightarrow a S b S b S \mid b S a S b S \mid b S b S a S \mid \lambda \)
   Provide a proof that shows
   \( L = \{ w \mid |w_b| = 2|w_a| \} \)
   That is, the number of b’s in \( w \) is twice that of the a’s
   You will need to provide an inductive proof in both directions
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 | x \in L and, for no non-null y does xy \in L \} and
  – min(L) = \{ x | x \in L and, for no proper prefix of x, y, does y \in L \}
• Describe the languages produced by max and min. for each of :
  – L1 = \{ a^i b^j c^k | k \leq i or k \leq j \} CFL
    • max(L1) = \{ a^i b^j c^k | k =\max(i, j) \} Non-CFL
    • min(L1) = \{ \lambda \} (string of length 0) Regular
  – L2 = \{ a^i b^j c^k | k > i or k > j \} CFL
    • max(L2) = \{ \} (empty) Regular
    • min(L2) = \{ a^i b^j c^k | 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
Formalization of PDA

- $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$
- $Q$ is finite set of states
- $\Sigma$ is finite input alphabet
- $\Gamma$ 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 $\Gamma_e$ 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
  – \(\gamma\) is contents of stack (leftmost symbol is top)

• Single step derivation is defined by
  – \([q, ax, Z\alpha] \rightarrow [p, x, \beta\alpha]\) if \(\delta(q, a, Z)\) contains \((p, \beta)\)

• Multistep derivation (\(\rightarrow^*\)) is 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

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

- Give just one state, this is essentially stateless, except for stack
Top Down Parsing by PDA

\[ E \rightarrow E + T | T \]
\[ T \rightarrow T * F | F \]
\[ F \rightarrow (E) | \text{Int} \]

- \( \delta(q,+,+) = \{(q,\lambda)\} \), \( \delta(q,\text{*,*}) = \{(q,\lambda)\} \),
- \( \delta(q,\text{Int,Int}) = \{(q,\lambda)\} \),
- \( \delta(q,\text{,}() = \{(q,\lambda)\}, \delta(q,\text{,)}) = \{(q,\lambda)\} \)
- \( \delta(q,\lambda,E) = \{(q,E+T), (q,T)\} \)
- \( \delta(q,\lambda,T) = \{(q,T*F), (q,F)\} \)
- \( \delta(q,\lambda,F) = \{(q,(E)), (q,\text{Int})\} \)
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
- $E(A) = \mathcal{L}(G)$
- Could also do $\delta(q,\lambda,S\$) \supseteq \{(q,\lambda)\}$, $N(A) = \mathcal{L}(G)$
Bottom Up Parsing by PDA

\[ E \rightarrow E + T | T \]
\[ T \rightarrow T * F | F \]
\[ F \rightarrow (E) | \text{Int} \]

\[ \delta(q,+,\lambda) = \{(q,+)\}, \quad \delta(q,*,\lambda) = \{(q,\ast)\}, \quad \delta(q,\text{Int},\lambda) = \{(q,\text{Int})\}, \]
\[ \delta(q,\text{(},\lambda) = \{(q,\text{)}\}, \quad \delta(q,\text{)},\lambda) = \{(q,\text{)})\} \]
\[ \delta(q,\lambda, T+\text{E}) = \{(q,\text{E})\}, \quad \delta(q,\lambda, T) \supseteq \{(q,\text{E})\} \]
\[ \delta(q,\lambda, F*T) \supseteq \{(q,\text{T})\}, \quad \delta(q,\lambda, F) \supseteq \{(q,\text{T})\} \]
\[ \delta(q,\lambda, \text{E}(\text{)}) \supseteq \{(q,\text{F})\}, \quad \delta(q,\lambda, \text{Int}) \supseteq \{(q,\text{F})\} \]
\[ \delta(q,\lambda, \text{E}) \supseteq \{(f,\lambda)\} \]
\[ \delta(f,\lambda, \$) = \{(f,\lambda)\} \]
\[ E(A) = \mathcal{L}(G) \]
Challenge

• Use the two recognizers on some sets of expressions like
  – $5 + 7 \times 2$
  – $5 \times 7 + 2$
  – $(5 + 7) \times 2$
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 $\{(q_0, Z\$)\}$
  – $\delta'$ does what $\delta$ 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 $\lambda$ 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
  – $V = \{S\} \cup \{<q, X, p>| q, p \in Q, X \in \Gamma\}$
  – $R$ on next page
Rules for PDA to CFG

• R contains rules as follows:
  S → <q₀,$,f> where F = {f}
  meaning: want to generate w whenever
  [q₀,w,$] |—*[f,λ,λ]

• Remaining rules are:
  <q,X,p> → a<s,Y,t><t,X,p>
  whenever δ(q,a,X) ⊇ {(s,PUSH(Y))}
  <q,X,p> → a
  whenever δ(q,a,X) ⊇ {(p,POP)}

• Want <q,X,p>⇒*w when [q,w,X] |—*[p,λ,λ]
Greibach Normal Form

- Each rule of a GNF is constrained to be of form: \( A \rightarrow a\alpha, \quad A \in V, \ a \in \Sigma, \ \alpha \in V^* \)
- If the language contains \( \lambda \) then we allow \( S \rightarrow \lambda \)
  and constrain \( S \) to not be on right hand side of any rule
- The beauty of this form is that, in a bottom up parse, every step consumes an input character and so parse is linear (if we guess right)
- We will not show details of conversion but it is dependent on starting in CNF and then removing left recursion, both of which we have already shown
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.

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 $A_2 = ( Q_0 \times Q_1, \Sigma, \Gamma, \delta_2, <q_0,q_1> \$, F_0 \times F_1)$ where

\[
\delta_2(<q,s>, a, X) \supseteq \{(q',s'), \alpha)\}, \ a \in \Sigma \cup \{\lambda\}, \ X \in \Gamma \iff \\
\delta_0(q, a, X) \supseteq \{(q', \alpha)\} \text{ and} \\
\delta_1(s, a) = s' \text{ (if } a = \lambda \text{ then } s' = s).\]

- Using the definition of derivations we see that
  \[
  [<q_0,q_1>, w, \$] \vdash^* [t,s>, \lambda, \beta] \text{ in } A_2 \iff \\
  [q_0, w, \$] \vdash^* [t, \lambda, \beta] \text{ in } A_0 \text{ and} \\
  [q_1, w] \vdash^* [s, \lambda] \text{ in } A_1
  \]
  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 $\Sigma$ 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 $R' = \{ A \to g(\alpha) \text{ where } A \to \alpha \text{ is in } R \}$ and consider only sentential forms over $\bigcup_{a \in \Sigma} S_a$, then $S \Rightarrow^* S_{a_1} S_{a_2} \ldots S_{a_n}$ in $G'$ iff $S \Rightarrow^* a_1 a_2 \ldots a_n$ in $G$ iff $a_1 a_2 \ldots a_n \in \mathcal{L}(G)$. 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.
Context Sensitive
Context Sensitive Grammar

G = (V, Σ, 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 → λ 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.
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 \text{\(//\) n > 0}

Cb \rightarrow bC \quad \text{\textit{// Shuttle C over to a c}
Cc \rightarrow cc \quad \text{\textit{// Change C 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

\[ L = \{ \text{ww} \mid w \in \{0,1\}^+ \} \]

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

\[ S \rightarrow 00 \mid 11 \mid 0A0<> \mid aA1<> \mid 1A1<> \]
\[ A \rightarrow 0AZ \mid 1AX \mid 0Z \mid 1X \]
\[ Z0 \rightarrow 0Z \quad Z1 \rightarrow 1Z \quad \text{// Shuttle Z (for owe zero)} \]
\[ X0 \rightarrow 0X \quad X1 \rightarrow 1X \quad \text{// Shuffle X (for owe one)} \]
\[ Z0<> \rightarrow 00<> \quad Z1<> \rightarrow 10<> \quad \text{// New 0 must be on rhs of old 0/1’s} \]
\[ X0<> \rightarrow 01<> \quad X1<> \rightarrow 11<> \quad \text{// New 1 must be on rhs of old 0/1’s} \]
\[ <0> \rightarrow 0 \quad \text{// Guess we are done} \]
\[ <1> \rightarrow 1 \quad \text{// Guess we are done} \]
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.
HISTORY

The Quest for Mechanizing Mathematics
Hilbert, Russell and Whitehead

• Until 1800’s there were no formal systems to reason about mathematical properties
• Major advances in late 1800’s/early 1900’s
• Axiomatic schemes
  – Axioms plus sound rules of inference
  – Much of focus on number theory
• First Order Predicate Calculus
  – $\forall x \exists y \ [y > x]$
• Second Order (Peano’s Axiom)
  – $\forall P \ [(P(0) \land \forall x [P(x) \implies P(x+1)]) \implies \forall x P(x)]$
Hilbert

• 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 zeros of polynomial equations with integral coefficients. This is now known to be impossible (In 1972, Matiyasevič showed this undecidable).
Hilbert’s Belief

• All mathematics could be developed within a formal system that allowed the mechanical creation and checking of proofs.
Gödel

- In 1931 he showed that any first order theory that embeds elementary arithmetic is either incomplete or inconsistent.
- He did this by showing that such a first order theory cannot reason about itself. That is, there is a first order expressible proposition that cannot be either proved or disproved, or the theory is inconsistent (some proposition and its complement are both provable).
- Gödel also developed the general notion of recursive functions but made no claims about their strength.
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.
• Kleene demonstrated the computational equivalence of recursively defined functions to Post-Turing machines.
• Church’s notation was the lambda calculus, which later gave birth to Lisp.
More on Emil Post

• In the 1920’s, starting with notation developed by Frege and others in 1880s, Post devised the truth table form we all use now for Boolean expressions (propositional logic). This was a part of his PhD thesis in which he showed the axiomatic completeness of the propositional calculus (all tautologies can be deduced from a finite set of tautologies and a finite set of rules of inference).

• In the late 1930’s and the 1940’s, Post devised symbol manipulation systems in the form of rewriting rules (precursors to Chomsky’s grammars). He showed their equivalence to Turing machines.

• In 1940s, Post showed the complexity (undecidability) of determining what is derivable from an arbitrary set of propositional axioms.
Computability

The study of what can/cannot be done via purely mechanical means
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 domain of possible inputs.
Sets and Decision Problems

- **Set** -- A collection of atoms from some universe $U$. $\emptyset$ denotes the empty set.
- **(Decision) Problem** -- A set of questions, each of which has answer “yes” or “no”.

© UCF EECS
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”).

• **Solved** -- A problem P is said to solved if P is solvable and we have produced its solution.

• **Unsolved, Unsolvable (Undecidable)** -- Complements of above
• **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 $\mathbb{N}$, 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 $\mathcal{P}$, 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

• $\mathcal{P}$ solved implies $\mathcal{P}$ solvable implies $\mathcal{P}$ semi-decidable (re, recognizable).
• $\mathcal{P}$ non-re implies $\mathcal{P}$ unsolvable implies $\mathcal{P}$ unsolved.
• $\mathcal{P}$ finite implies $\mathcal{P}$ solvable.
Slightly Harder Implications

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

- We will prove these later.
Existence of Undecidables

• A counting argument
  – The number of mappings from $\mathbb{N}$ to $\mathbb{N}$ is at least as great as the number of subsets of $\mathbb{N}$. But the number of subsets of $\mathbb{N}$ 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 (In 1970, Matiyasevič showed this undecidable).
Hilbert’s 10\textsuperscript{th} is Semi-Decidable

- Consider over one variable: \( P(x) = 0 \)
- Can semi-decide by plugging in 
  0, 1, -1, 2, -2, 3, -3, …
- 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 \( P(x_1,x_2,\ldots,x_k) = 0 \).
P(x) = 0 is Decidable

- $c_n x^n + c_{n-1} x^{n-1} + \ldots + c_1 x + c_0 = 0$
- $x^n = -(c_{n-1} x^{n-1} + \ldots + c_1 x + c_0)/c_n$
- $|x^n| \leq c_{\text{max}}(|x^{n-1}| + \ldots + |x| + 1)/|c_n|$
- $|x^n| \leq c_{\text{max}}(n \cdot |x^{n-1}|)/|c_n|$, since $|x| \geq 1$
- $|x| \leq n \times c_{\text{max}}/|c_n|$
P(x) = 0 is Decidable

• Can bound the search to values of x in range \([±n \ast (c_{\text{max}} / c_n )]\), where
  - n = highest order exponent in polynomial
  - \(c_{\text{max}}\) = largest absolute value coefficient
  - \(c_n\) = coefficient of highest order term

• Once we have a search bound and we are dealing with a countable set, we have an algorithm to decide if there is an x.

• Cannot find bound when more than one variable, so cannot extend to \(P(x_1,x_2,..,x_k) = 0\).
Undecidability

We Can’t Do It All
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." 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?

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 \( \{ y \mid f(x)\downarrow \text{ and } f(x) = y \} \).
Assume we can decide the halting problem. Then there exists some total function $Halt$ such that

$$Halt(x, y) = \begin{cases} 1 & \text{if } \varphi_x(y) \downarrow \\ 0 & \text{if } \varphi_x(y) \uparrow \end{cases}$$

Here, we have numbered all programs and $\varphi_x$ refers to the $x$-th program in this ordering. Now we can view $Halt$ as a mapping from $\mathbb{N}$ into $\mathbb{N}$ by treating its input as a single number representing the pairing of two numbers via the one-one onto function

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

with inverses

$$<z>_1 = \log_2(z+1)$$

$$<z>_2 = ((( z + 1 ) // 2 <z>_1 ) - 1 ) // 2$$
The Contradiction

Now if $\text{Halt}$ exist, then so does $\text{Disagree}$, where

$$\begin{align*}
\text{Disagree}(x) &= 0 & \text{if } \text{Halt}(x,x) = 0, \text{ i.e., if } \varphi_x(x) \uparrow \\
&= \mu y \ (y = y+1) & \text{if } \text{Halt}(x,x) = 1, \text{ i.e., if } \varphi_x(x) \downarrow
\end{align*}$$

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

$\text{Disagree}(d)$ is defined $\iff \text{Halt}(d,d) = 0$

$\iff \varphi_d(d) \uparrow$

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

But this means that $\text{Disagree}$ 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 is not solvable.
Halting is recognizable

While the Halting Problem is not solvable, it is re, 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.

```plaintext
Semi_Decide_Halting() {
    Read P, x;
    P(x);
    Print “yes”;
}
```
Why not just algorithms?

A question that might come to mind is why we could not just have a model of computation that involves only programs that halt for all input. Assume you have such a model – our claim is that this model must be incomplete!

Here’s the logic. 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 \( \phi_x \) is the \( x \)-th procedure and \( \phi_x(y) \) is the \( x \)-th procedure with input \( y \). We also refer to \( x \) as the procedure’s index.
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 $\text{Univ}$. 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 $\text{re}$, we will assume both arguments are natural numbers; the first being the index of the program. Thus,

$$\text{Univ}(x,y) = \varphi_x(y)$$
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 algorithms can be enumerated, and that $F$ accomplishes this. Then

\[ F(x) = F_x \]

where $F_0, F_1, F_2, \ldots$ 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.
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 just simple finite iteration or recursion?

• Fixed Point: Ordered Petri Nets, (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
Models of Computation

Turing Machines
Register Machines
Factor Replacement Systems
Recursive Functions
Turing Machines

1st Model

A Linear Memory Machine
Typical Textbook Description

- A Turing machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$
- $Q$ is a finite set of states
- $\Sigma$, is a finite input alphabet not containing the blank symbol $\sqcup$
- $\Gamma$ is a finite set of tape symbols that includes $\Sigma$ and $\sqcup$ commonly $\Gamma = \Sigma \cup \{\sqcup\}$
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{R,L\}$
- $q_0$ starts, $q_{\text{accept}}$ accepts, $q_{\text{reject}}$ rejects
The Turing description just given requires you to write a new symbol and move off the current tape square.

Post had a variant where \(\delta: Q \times \Gamma \rightarrow Q \times (\Gamma \cup \{R,L\})\).

Here, you either write or move, not both.

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 create 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 arc.
• 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

$R$ -- move right to next 0 (not including current square)

$\ldots ?11\ldots 10\ldots \Rightarrow \ldots ?11\ldots 10\ldots$

$L$ -- move left to next 0 (not including current square)

$\ldots 011\ldots 1?\ldots \Rightarrow \ldots 011\ldots 1?\ldots$
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, $x_1, \ldots, x_n$ in the form

$$\ldots 01^{x_1}01^{x_2}0\ldots 01^{x_n}0\ldots$$

and ends with

$$\ldots 01^{x_1}01^{x_2}0\ldots 01^{x_n}01^y0\ldots$$

where $y = F(x_1, \ldots, x_n)$. 
Need the copy family of useful submachines, where $C_k$ copies $k$-th preceding value.

The add machine is then

$$
\begin{array}{cccccc}
\mathcal{L}^k & R & 0 & \mathcal{R}^k & 0 & \mathcal{R}^{k+1} & 1 & \mathcal{L}^{k+1} & 1 \\
& 1 & & & & & & & \\
\end{array}
$$

The add machine is then

$$
C_2 \ C_2 \ \mathcal{L} \ 1 \ \mathcal{R} \ \mathcal{L} \ 0
$$
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 registers 1 to $n$ and ends with the result in the 0-th 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 $k$-th register, for any $k$, such that there are at least $k+n+1$ registers.
Register Instructions

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

  \[ \text{INC}_r[i] \] –
  increment \( r \) and jump to \( i \).

  \[ \text{DEC}_r[p, z] \] –
  if register \( r > 0 \), decrement \( r \) and jump to \( p \)
  else jump to \( z \)

• Note, we do not use subscripts if obvious.
Addition by RM

Addition (r0 ← r1 + r2)
1. DEC0[1,2] : Zero result (r0) and work (r3) registers
2. DEC3[2,3]
3. DEC1[4,6] : Add r1 to r0, saving original r1 in r3
4. INC0[5]
5. INC3[3]
6. DEC3[7,8] : Restore r1
7. INC1[6]
8. DEC2[9,11] : Add r2 to r0, saving original r2 in r3
9. INC0[10]
10. INC3[8]
11. DEC3[12,13] : Restore r2
13. : Halt by branching here

In many cases we just assume registers, other those with input, are zero at start. That would remove need instructions 1 and 2.
Limited Subtraction by RM

Subtraction (r0 ← r1 - r2, if r1\geq r2; 0, otherwise)
1. DEC0[1,2]: Zero result (r0) and work (r3) registers
2. DEC3[2,3]
3. DEC1[4,6]: Add r1 to r0, saving original r1 in r3
4. INC0[5]
5. INC3[3]
6. DEC3[7,8]: Restore r1
7. INC1[6]
8. DEC2[9,11]: Subtract r2 from r0, saving original r2 in r3
9. DEC0[10,10]: Note that decrementing 0 does nothing
10. INC3[8]
11. DEC3[12,13]: Restore r2
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 \( x \).

• A fraction \( \frac{a}{b} \) is applicable to some natural number \( x \), just in case \( x \) is divisible by \( b \). We always chose the first applicable fraction \( \frac{a}{b} \), multiplying it times \( x \) to produce a new natural number \( x \cdot \frac{a}{b} \). The process is then applied to this new number.

• Termination occurs when no fraction is applicable.

• A factor replacement system partially computing \( n \)-ary function \( F \) typically starts with its argument encoded as powers of the first \( n \) odd primes. Thus, arguments \( x_1, x_2, \ldots, x_n \) are encoded as \( 3^{x_1}5^{x_2}\ldots p_n^{x_n} \). The result then appears as the power of the prime 2.
Addition by FRS

Addition is $3^x15^x2$ becomes $2^x1+x^2$

or, in more details, $2^03^x15^x2$ becomes $2^x1+x^23^05^0$

$2 / 3$

$2 / 5$

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

$bx \rightarrow ax$

meaning that a number that has can be factored as $bx$ can have the factor $b$ replaced by an $a$.

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 $2^{\max(0,x_1-x_2)}$

$3\cdot5x \rightarrow x$
$3x \rightarrow 2x$
$5x \rightarrow x$
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 \( 135 = 3^35^1 \), deterministically we get

\[ 135 \Rightarrow 9 \Rightarrow 6 \Rightarrow 4 = 2^2 \]

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

\[ 135 \Rightarrow 9 \Rightarrow 6 \Rightarrow 4 = 2^2 \]
\[ 135 \Rightarrow 90 \Rightarrow 60 \Rightarrow 40 \Rightarrow 8 = 2^3 \]
\[ 135 \Rightarrow 45 \Rightarrow 3 \Rightarrow 2 = 2^1 \]
\[ 135 \Rightarrow 45 \Rightarrow 15 \Rightarrow 1 = 2^0 \]
\[ 135 \Rightarrow 45 \Rightarrow 15 \Rightarrow 5 \Rightarrow 1 = 2^0 \]
\[ 135 \Rightarrow 45 \Rightarrow 15 \Rightarrow 3 \Rightarrow 2 = 2^1 \]
\[ 135 \Rightarrow 45 \Rightarrow 9 \Rightarrow 6 \Rightarrow 4 = 2^2 \]
\[ 135 \Rightarrow 90 \Rightarrow 60 \Rightarrow 40 \Rightarrow 8 = 2^3 \]

\[ \ldots \]

This computes \( 2^z \) where \( 0 \leq z \leq x_1 \). 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 R1=x at starts; at termination, set R0=1 if x is odd; 0 otherwise. We assume R0=0 at start. We also are not concerned about destroying input.

1. DEC1[2, 4]
2. DEC1[1, 3]
3. INC0[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.

$3^3 x \rightarrow x$
$3 x \rightarrow 2 x$
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

$3^2 5 \rightarrow 5 7 x$

$3^1 5 7 \rightarrow x$

$3^1 5 \rightarrow 2 x$

$5 7 \rightarrow 7 11 x$

$7 11 \rightarrow 3 11 x$

$11 \rightarrow 5 x$

$5 x \rightarrow x$

$7 x \rightarrow x$
Systems Related to FRS

• **Petri Nets:**
  – Unordered
  – Ordered
  – Negated Arcs

• **Vector Addition Systems:**
  – Unordered
  – Ordered

• **Factors with Residues:**
  – \( a x + c \rightarrow b x + d \)

• **Finitely Presented Abelian Semi-Groups**
Petri Net Operation

- Finite number of places, each of which can hold zero or more markers.
- Finite number of transitions, each of which has a finite number of input and output arcs, starting and ending, respectively, at places.
- A transition is enabled if all the nodes on its input arcs have at least as many markers as arcs leading from them to this transition.
- Progress is made whenever at least one transition is enabled. Among all enabled, one is chosen randomly to fire.
- Firing a transition removes one marker per arc from the incoming nodes and adds one marker per arc to the outgoing nodes.
Petri Net Computation

• A Petri Net starts with some finite number of markers distributed throughout its $n$ nodes.
• The state of the net is a vector of $n$ natural numbers, with the $i$-th component’s number indicating the contents of the $i$-th node. E.g., $<0,1,4,0,6>$ could be the state of a Petri Net with 5 places, the 2nd, 3rd and 5th, having 1, 4, and 6 markers, resp., and the 1st and 4th being empty.
• Computation progresses by selecting and firing enabled transitions. Non-determinism is typical as many transitions can be simultaneously enabled.
• Petri nets are often used to model coordination algorithms, especially for computer networks.
Variants of Petri Nets

- A Petri Net is not computationally complete. In fact, its halting and word problems are decidable. However, its containment problem (are the markings of one net contained in those of another?) is not decidable.
- A Petri net with prioritized transitions, such that the highest priority transitions is fired when multiple are enabled is equivalent to an FRS. (Think about it).
- A Petri Net with negated input arcs is one where any arc with a slash through it contributes to enabling its associated transition only if the node is empty. These are computationally complete. They can simulate register machines. (Think about this also).
Petri Net Example

Diagram showing a Petri Net with places, transitions, markers, and arcs. The places are labeled with "Want R", "Want S", and "Release". Transitions are marked with "R" and "S". Arrows indicate the flow of tokens between places and transitions.
Vector Addition

• Start with a finite set of vectors in integer n-space.
• Start with a single point with non-negative integral coefficients.
• Can apply a vector only if the resultant point has non-negative coefficients.
• Choose randomly among acceptable vectors.
• This generates the set of reachable points.
• Vector addition systems are equivalent to Petri Nets.
• If order vectors, these are equivalent to FRS.
Vectors as Resource Models

• Each component of a point in $\mathbb{n}$-space represents the quantity of a particular resource.
• The vectors represent processes that consume and produce resources.
• The issues are safety (do we avoid bad states) and liveness (do we attain a desired state).
• Issues are deadlock, starvation, etc.
Factors with Residues

• Rules are of form
  \[ a_i x + c_i \rightarrow b_i x + d_i \]
  – There are \( n \) such rules
  – Can apply if number is such that you get a residue (remainder) \( c_i \) when you divide by \( a_i \)
  – Take quotient \( x \) and produce a new number \( b_i x + d_i \)
  – Can apply any applicable one (no order)

• These systems are equivalent to Register Machines.
Abelian Semi-Group

\( S = (G, \cdot) \) is a semi-group if

- \( G \) is a set, \( \cdot \) is a binary operator, and
- 1. Closure: If \( x, y \in G \) then \( x \cdot y \in G \)
- 2. Associativity: \( x \cdot (y \cdot z) = (x \cdot y) \cdot z \)

\( S \) is a monoid if

- 3. Identity: \( \exists e \in G \ \forall x \in G \ [e \cdot x = x \cdot e = x] \)

\( S \) is a group if

- 4. Inverse: \( \forall x \in G \ \exists x^{-1} \in G \ [x^{-1} \cdot x = x \cdot x^{-1} = e] \)

\( S \) is Abelian if \( \cdot \) is commutative
Finitely Presented

• $S = (G, \cdot)$, a semi-group (monoid, group), is finitely presented if there is a finite set of symbols, $\Sigma$, called the alphabet or generators, and a finite set of equalities ($\alpha_i = \beta_i$), the reflexive transitive closure of which determines equivalence classes over $G$.

• Note, the set $G$ is the closure of the generators under the semi-group’s operator $\cdot$.

• The problem of determining membership in equivalence classes for finitely presented Abelian semi-groups is equivalent to that of determining mutual derivability in an unordered FRS or Vector Addition System with inverses for each rule.
Recursive Functions

Primitive and $\mu$-Recursive
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:
  
  \[ C_a(x_1, \ldots, x_n) = a \] 
  : constant functions

  \[ I_i^n(x_1, \ldots, x_n) = x_i \] 
  : identity functions
  : aka projection

  \[ S(x) = x + 1 \] 
  : an increment function
Building New Functions

• **Composition:**
  
  If \( G, H_1, \ldots, H_k \) are already known to be primitive recursive, then so is \( F \), where
  \[
  F(x_1, \ldots, x_n) = G(H_1(x_1, \ldots, x_n), \ldots, H_k(x_1, \ldots, x_n))
  \]

• **Iteration (aka primitive recursion):**
  
  If \( G, H \) are already known to be primitive recursive, then so is \( F \), where
  \[
  F(0, x_1, \ldots, x_n) = G(x_1, \ldots, x_n)
  \]
  \[
  F(y+1, x_1, \ldots, x_n) = H(y, x_1, \ldots, x_n, F(y, x_1, \ldots, x_n))
  \]
  
  We also allow definitions like the above, except iterating on \( y \) as the last, rather than first argument.
Addition & Multiplication

Example: Addition

\[ + (0, y) = I_{1}(y) \]
\[ + (x+1, y) = H(x, y, +(x, y)) \]
where \( H(a, b, c) = S(I_{3}(a, b, c)) \)

Example: Multiplication

\[ * (0, y) = C_{0}(y) \]
\[ * (x+1, y) = H(x, y, *(x, y)) \]
where \( H(a, b, c) = +(I_{2}(a, b, c), I_{3}(a, b, c)) \)
\[ = b + c = y + *(x, y) = (x+1)*y \]
Basic Arithmetic

\[
x + 1:
\]
\[
x + 1 = S(x)
\]
\[
x - 1:
\]
\[
0 - 1 = 0
\]
\[
(x + 1) - 1 = x
\]
\[
x + y:
\]
\[
x + 0 = x
\]
\[
x + (y + 1) = (x + y) + 1
\]
\[
x - y:  \quad // \text{limited subtraction}
\]
\[
x - 0 = x
\]
\[
x - (y + 1) = (x - y) - 1
\]
2nd Grade Arithmetic

\[
x \times y:
\]
\[
x \times 0 = 0
\]
\[
x \times (y+1) = x \times y + x
\]

\[
x!:
\]
\[
0! = 1
\]
\[
(x+1)! = (x+1) \times x!
\]
Basic Relations

x == 0:
   0 == 0 = 1
   (y+1) == 0 = 0

x == y:
   x==y = ((x – y) + (y – x )) == 0

x ≤y :
   x≤y = (x – y) == 0

x ≥ y:
   x≥y = y≤x

x > y :
   x>y = ~(x≤y) /* See ~ on next page */

x < y :
   x<y = ~(x≥y)
Basic Boolean Operations

\( \sim x: \)
\[ \sim x = 1 - x \text{ or } (x==0) \]

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

\[ f(x) = \begin{cases} 
  g(x) & \text{if } P(x) \\
  h(x) & \text{otherwise} 
\end{cases} \]

\[ f(x) = P(x) \cdot g(x) + (1-P(x)) \cdot h(x) \]

Can use induction to prove this is true for all \( k>0 \), where

\[ f(x) = \begin{cases} 
  g_1(x) & \text{if } P_1(x) \\
  g_2(x) & \text{if } P_2(x) \& \& \neg P_1(x) \\
  \vdots \\
  g_k(x) & \text{if } P_k(x) \& \& \neg(P_1(x) \| \ldots \| \neg P_{k-1}(x)) \\
  h(x) & \text{otherwise} 
\end{cases} \]
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

\[ f(0) \quad = \quad 1 - P(0) \]

\[ f(x+1) \quad = \quad f(x) \quad \text{if} \quad f(x) \leq x \]

\[ = \quad x + 2 - P(x+1) \quad \text{otherwise} \]
Bounded Minimization 2

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

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

\( x \parallel y:\)
\[
\begin{align*}
  x \parallel 0 &= 0 & \text{: silly, but want a value} \\
  x \parallel (y+1) &= \mu z \ (z < x) \ [ \ (z+1)(y+1) > x ]
\end{align*}
\]

\( x \mid y:\) \( x \) is a divisor of \( y \)
\[
\begin{align*}
  x \mid y &= ((y \parallel x) \ast x) == y
\end{align*}
\]
Primality

\textbf{firstFactor}(x): first non-zero, non-one factor of \(x\).

\[
\text{firstfactor}(x) = \mu z \ (2 \leq z \leq x) \ [z|x],
\]

0 if none

\textbf{isPrime}(x):

\[
isPrime(x) = \text{firstFactor}(x) == x \land (x>1)
\]

\textbf{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 \text{prime}(i)
Exponents

\[^{y}x:\]
\[x^0 = 1\]
\[x^{(y+1)} = x \times x^y\]

\[\exp(x,i):\] the exponent of \(p_i\) in number \(x\).
\[\exp(x,i) = \mu \ z \ (z<x) \ [ \sim(p_i^{z+1} | x) \ ]\]
Pairing Functions

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

• with inverses
  \( \langle z \rangle_1 = \exp(z+1,0) \)

  \( \langle z \rangle_2 = ((( z + 1 ) \mod 2 \langle z \rangle_1 ) - 1 ) \mod 2 \)

• These are very useful and can be extended to encode \( n \)-tuples
  \( \langle x, y, z \rangle = \langle x, \langle y, z \rangle \rangle \) (note: stack analogy)
Prove that the pairing function \(<x,y> = 2^x (2y + 1) - 1\) 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, \(<x,y>+1\), which implies the problem of mapping the pairing function to \(Z^+\).
Case 1 (x=0)

Case 1:
For \( x = 0 \), \( <0,y>+1 = 2^0(2y+1) = 2y+1 \). 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 when \( x=0 \). Thus, \( <0,y>+1 \) is 1-1 onto the odd natural numbers.
Case 2 (x > 0)

Case 2:
For x > 0, <x,y>+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^xz, where x>0, z is an odd number and this pair x,y is unique. Thus, <x,y>+1 is 1-1 onto the even natural numbers, when x>0.

The above shows that <x,y>+1 is 1-1 onto Z^+, but then <x,y> is 1-1 onto N, as was desired.
Pairing Function is 1-1 Onto

Approach 2:
Another approach to show a function $f$ over $S$ is 1-1 onto $T$ is to show that $f^{-1}(f(x)) = x$, for arbitrary $x \in S$ and that $f(f^{-1}(z)) = z$, for arbitrary $z \in T$.

Thus, we need to show that $(<x,y>_1,<x,y>_2) = (x,y)$ for arbitrary $(x,y) \in \mathbb{N} \times \mathbb{N}$ and $<<z>_1,<z>_2> = z$ for arbitrary $z \in \mathbb{N}$. 
Alternate Proof

Let \(x, y\) be arbitrary natural numbers, then \(<x, y> = 2^x(2y+1)-1\).

Moreover, \(<2^x(2y+1)-1,>_1 = \text{Factor}(2^x(2y+1), 0) = x\), since \(2y+1\) must be odd, and

\(<2^x(2y+1)-1,>_2 = ((2^x(2y+1)/2^{\text{Factor}(2^x(2y+1), 0)})-1)/2 = 2y/2 = y\).

Thus, \((<x, y>_1, <x, y>_2) = (x, y)\), as was desired.

Let \(z\) be an arbitrary natural number, then the inverse of the pairing is \((<z>_1, <z>_2)\)

Moreover, \(<<z>_1, <z>_2> = 2^{<z>_1} * (2^{<z>_2+1}-1)\)

\(= 2^{\text{Factor}(z+1, 0)} * (2^z(z+1)/ 2^{\text{Factor}(z+1, 0)})/2-1+1)-1\)

\(= 2^{\text{Factor}(z+1, 0)} (z+1)/ 2^{\text{Factor}(z+1, 0)})-1\)

\(= (z+1) – 1\)

\(= z\), as was desired.
Application of Pairing

Show that prfs are closed under Fibonacci induction. Fibonacci induction means that each induction step after calculating the base is computed using the previous two values, where the previous values for f(1) are f(0) and 0; and for x>1, f(x) is based on f(x-1) and f(x-2).

The formal hypothesis is:
Assume g and h are already known to be prf, then so is f, where
f(0,x) = g(x);
f(1,x) = h(f(0,x), 0); and
f(y+2,x) = h(f(y+1,x), f(y,x))

Proof is by construction
Fibonacci Recursion

Let K be the following primitive recursive function, defined by induction on the primitive recursive functions, g, h, and the pairing function.

\[
K(0,x) = B(x)
\]

\[
B(x) = \langle g(x), C_0(x) \rangle \quad \text{// this is just } \langle g(x), 0 \rangle
\]

\[
K(y+1, x) = J(y, x, K(y,x))
\]

\[
J(y,x,z) = \langle h(<z>_1, <z>_2), <z>_1 \rangle \quad \text{// this is } \langle f(y+1,x), f(y,x) \rangle, \text{ even though f is not yet shown to be prf!!}
\]

This shows K is prf.

f is then defined from K as follows:

\[
f(y,x) = \langle K(y,x) \rangle_1 \quad \text{// extract first value from pair encoded in } K(y,x)
\]

This shows it is also a prf, 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

- $A(1, j)=2j$ 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.
Union/Find

• Start with a collection $S$ of unrelated elements – singleton equivalence classes
• $\text{Union}(x,y)$, $x$ and $y$ are in $S$, merges the class containing $x$ ($[x]$) with that containing $y$ ($[y]$)
• $\text{Find}(x)$ returns the canonical element of $[x]$
• Can see if $x \equiv y$, by seeing if $\text{Find}(x) == \text{Find}(y)$
• How do we represent the classes?
The $\mu$ Operator

- Minimization:
  If $G$ is already known to be recursive, then so is $F$, where
  $$F(x_1,\ldots,x_n) = \mu y \ (G(y,x_1,\ldots,x_n) == 1)$$

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

Equivalency of computation by Turing machines, register machines, factor replacement systems, recursive functions
Proving Equivalence

• Constructions do not, by themselves, prove equivalence.
• To do so, we need to develop a notion of an “instantaneous description” (id) of each model of computation (well, almost as recursive functions are a bit different).
• We then show a mapping of id’s between the models.
Instantaneous Descriptions

- An instantaneous description (id) is a finite description of a state achievable by a computational machine, \(M\).
- Each machine starts in some initial id, \(id_0\).
- The semantics of the instructions of \(M\) define a relation \(\Rightarrow_M\) such that, \(id_i \Rightarrow_M id_{i+1}, i \geq 0\), if the execution of a single instruction of \(M\) would alter \(M\)'s state from \(id_i\) to \(id_{i+1}\) or if \(M\) halts in state \(id_i\) and \(id_{i+1} = id_i\).
- \(\Rightarrow^+_M\) is the transitive closure of \(\Rightarrow_M\)
- \(\Rightarrow^*_M\) is the reflexive transitive closure of \(\Rightarrow_M\)
id Definitions

• For a register machine, $M$, an id is an $s+1$ tuple of the form $(i, r_1, \ldots, r_s)_M$ specifying the number of the next instruction to be executed and the values of all registers prior to its execution.
• For a factor replacement system, an id is just a natural number.
• For a Turing machine, $M$, an id is some finite representation of the tape, the position of the read/write head and the current state. This is usually represented as a string $\alpha q x \beta$, where $\alpha$ ($\beta$) is the shortest string representing all non-blank squares to the left (right) of the scanned square, $x$ is the symbol at the scanned square and $q$ is the current state.
• Recursive functions do not have id’s, so we will handle their simulation by an inductive argument, using the primitive functions are the basis and composition, induction and minimization in the inductive step.
Equivalence Steps

- Assume we have a machine $M$ in one model of computation and a mapping of $M$ into a machine $M'$ in a second model.
- Assume the initial configuration of $M$ is $id_0$ and that of $M'$ is $id'_0$.
- Define a mapping, $h$, from id’s of $M$ into those of $M'$, such that, $R_M = \{ h(d) | d \text{ is an instance of an id of } M \}$, and
  - $id'_0 \Rightarrow^{*}_{M'} h(id_0)$, and $h(id_0)$ is the only member of $R_M$ in the configurations encountered in this derivation.
  - $h(id_i) \Rightarrow^{+}_{M'} h(id_{i+1})$, $i \geq 0$, and $h(id_{i+1})$ is the only member of $R_M$ in this derivation.
- The above, in effect, provides an inductive proof that
  - $id_0 \Rightarrow^{*}_M id$ implies $id'_0 \Rightarrow^{*}_{M'} h(id)$, and
  - If $id'_0 \Rightarrow^{*}_{M'} id'$ then either $id_0 \Rightarrow^{*}_M id$, where $id' = h(id)$, or $id' \not\in R_M$.
All Models are Equivalent

Equivalency of computation by Turing machines, register machines, factor replacement systems, recursive functions
Our Plan of Attack

• We will now show
\[ \text{TURING} \leq \text{REGISTER} \leq \text{FACTOR} \leq \text{RECURSIVE} \leq \text{TURING} \]
where by \( A \leq B \), we mean that every instance of \( A \) can be replaced by an equivalent instance of \( B \).

• The transitive closure will then get us the desired result.
TURING $\leq$ REGISTER
Encoding a TM’s State

- Assume that we have an \( n \) state Turing machine. Let the states be numbered 0,..., \( n-1 \).
- Assume our machine is in state 7, with its tape containing
  \[ ... 0 0 1 0 1 0 0 1 1 \_ q7 0 0 0 ... \]
- The underscore indicates the square being read. We denote this by the finite id
  \[ 1 0 1 0 0 1 1 q7 0 \]
- In this notation, we always write down the scanned square, even if it and all symbols to its right are blank.
More on Encoding of TM

- An id can be represented by a triple of natural numbers, \((R, L, i)\), where \(R\) is the number denoted by the reversal of the binary sequence to the right of the \(q_i\), \(L\) is the number denoted by the binary sequence to the left, and \(i\) is the state index.

- So,
  \[
  \ldots 0 0 1 0 1 0 0 1 1 \quad q7 \quad 0 0 0 \ldots
  \]
  is just \((0, 83, 7)\).

  \[
  \ldots 0 0 1 0 \quad q5 \quad 1 0 1 1 0 0 \ldots
  \]
  is represented as \((13, 2, 5)\).

- We can store the \(R\) part in register 1, the \(L\) part in register 2, and the state index in register 3.
Simulation by RM

1. DEC3[2,q0] : Go to simulate actions in state 0
2. DEC3[3,q1] : Go to simulate actions in state 1
...
n. DEC3[ERR,qn-1] : Go to simulate actions in state n-1
...
qj. IF_r1_ODD[qj+2] : Jump if scanning a 1
qj+1. JUMP[set_k] : If (qj 0 0 qk) is rule in TM
qj+1. INC1[set_k] : If (qj 0 1 qk) is rule in TM
qj+1. DIV_r1_BY_2 : If (qj 0 R qk) is rule in TM
   MUL_r2__BY_2
   JUMP[set_k]
qj+1. MUL_r1_BY_2 : If (qj 0 L qk) is rule in TM
   IF_r2_ODD then INC1
   DIV_r2__BY_2[set_k]
...
set_n-1. INC3[set_n-2] : Set r3 to index n-1 for simulating state n-1
set_n-2. INC3[set_n-3] : Set r3 to index n-2 for simulating state n-2
...
set_0. JUMP[1] : Set r3 to index 0 for simulating state 0
Fixups

• Need epilog so action for missing quad (halting) jumps beyond end of simulation to clean things up, placing result in r0.
• Can also have a prolog that starts with arguments in registers r1 to rn and stores values in r1, r2 and r3 to represent Turing machines starting configuration.
Example assuming \( n \) arguments (fix as needed)

1. \( \text{MUL}_r\text{n+1}_\text{BY}_2[2] \): Set \( r_{n+1} = 11\ldots10 \), where, \#1's = \( r_1 \)
2. \( \text{DEC}_1[3,4] \) : \( r_1 \) will be set to 0
3. \( \text{INC}_{n+1}[1] \) :
4. \( \text{MUL}_r\text{n+1}_\text{BY}_2[5] \): Set \( r_{n+1} = 11\ldots1011\ldots10 \), where, \#1's = \( r_1 \), then \( r_2 \)
5. \( \text{DEC}_2[6,7] \) : \( r_2 \) will be set to 0
6. \( \text{INC}_{n+1}[4] \) :

\[ \ldots \]

3\( n-2 \). \( \text{DEC}_n[3n-1,3n+1] \) : Set \( r_{n+1} = 11\ldots1011\ldots1011\ldots10 \), where, \#1's = \( r_1, r_2, \ldots \)
3\( n-1 \). \( \text{MUL}_r\text{n+1}_\text{BY}_2[3n] \) : \( r_n \) will be set to 0
3\( n \). \( \text{INC}_n[1][3n-2] \) :
3\( n+1 \) \( \text{DEC}_n[1][3n+2,3n+3] \) : Copy \( r_{n+1} \) to \( r_2 \), \( r_{n+1} \) is set to 0
3\( n+2 \). \( \text{INC}_2[3n+1] \) :
3\( n+3 \). \( \text{DEC}_n[1][3n+2,3n+3] \) : \( r_2 = \) left tape, \( r_1 = 0 \) (right), \( r_3 = 0 \) (initial state)
Epilog

1. DEC3[1,2] : Set r3 to 0 (just cleaning up)
2. IF_r1_ODD[3,5] : Are we done with answer?
3. INC0[4] : putting answer in r0
4. DIV_r1_BY_2[2] : strip a 1 from r1
5. : Answer is now in r0
REGISTER ≤ FACTOR
Encoding a RM’s State

• This is a really easy one based on the fact that every member of $\mathbb{Z}^+$ (the positive integers) has a unique prime factorization. Thus all such numbers can be uniquely written in the form

$$p_{i_1}^{k_1} p_{i_2}^{k_2} \cdots p_{i_j}^{k_j}$$

where the $p_i$'s are distinct primes and the $k_i$'s are non-zero values, except that the number 1 would be represented by $2^0$.

• Let $R$ be an arbitrary $n+1$-register machine, having $m$ instructions.

Encode the contents of registers $r_0, \ldots, r_n$ by the powers of $p_0, \ldots, p_n$.

Encode rule number's $1, \ldots, m$ by primes $p_{n+1}, \ldots, p_{n+m}$

Use $p_{n+m+1}$ as prime factor that indicates simulation is done.

• This is, in essence, a Gödel number of the RM’s state.
Simulation by FRS

• Now, the j-th instruction \(1 \leq j \leq m\) of \(R\) has associated factor replacement rules as follows:
  j. \text{INCr}[i]
      \[ p_{n+j}x \rightarrow p_{n+i}p_rx \]
  j. \text{DECr}[s, f]
      \[ p_{n+j}p_rx \rightarrow p_{n+s}x \]
      \[ p_{n+j}x \rightarrow p_{n+f}x \]

• We also add the halting rule associated with \(m+1\) of
  \[ p_{n+m+1}x \rightarrow x \]
Importance of Order

• The relative order of the two rules to simulate a DEC are critical.

• To test if register r has a zero in it, we, in effect, make sure that we cannot execute the rule that is enabled when the r-th prime is a factor.

• If the rules were placed in the wrong order, or if they weren't prioritized, we would be non-deterministic.
Sample RM and FRS (repeat)

Present a Register Machine that computes IsOdd. Assume R1=x at starts; at termination, set R0=1 if x is odd; 0 otherwise. We assume R0=0 at start. We also are not concerned about destroying input.

1. DEC1[2, 4]
2. DEC1[1, 3]
3. INC0[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.

$3*3 \ x \rightarrow \ x$

$3 \ x \rightarrow \ 2 \ x$
Example of Order

Consider the simple machine to compute
\( r_0 := r_1 - r_2 \) (limited)

1. DEC2[2,3]
2. DEC1[1,1]
3. DEC1[4,5]
4. INC0[3]
5.
Subtraction Encoding

Start with $3x5y7$

- $7 \cdot 5 x \rightarrow 11 x$
- $7 x \rightarrow 13 x$
- $11 \cdot 3 x \rightarrow 7 x$
- $11 x \rightarrow 7 x$
- $13 \cdot 3 x \rightarrow 17 x$
- $13 x \rightarrow 19 x$
- $17 x \rightarrow 13 \cdot 2 x$
- $19 x \rightarrow x$
Analysis of Problem

• If we don't obey the ordering here, we could take an input like $3^55^27$ and immediately apply the second rule (the one that mimics a failed decrement).

• We then have $3^55^213$, signifying that we will mimic instruction number 3, never having subtracted the 2 from 5.

• Now, we mimic copying r1 to r0 and get $2^55^219$.

• We then remove the 19 and have the wrong answer.
FACTOR ≤ RECURSIVE
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 \((n, ((a_1,b_1), (a_2,b_2), \ldots ,(a_n,b_n))\) be some factor replacement system, where \((a_i,b_i)\) means that the \(i\)-th rule is \(a_i \times \mathbb{F} \rightarrow b_i \times \mathbb{F}\).

• Encode this machine by the number \(F_n = \frac{\sum \left(2^n \cdot a_i \cdot 3^i \cdot b_i \right)}{\prod p_i}\), where \(p_i\) are prime numbers.

\[2^n a_1 5^{b_1} 7 a_2 11^{b_2} \ldots p_{2n-1}^{a_n} p_{2n}^{b_n} p_{2n+1} p_{2n+2}^{a_n} \]
Simulation by Recursive # 1

• We can determine the rule of $F$ that applies to $x$ by

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

• Note: $\exp(F, 2*i-1) = a_i$ where $a_i$ is the exponent of the prime factor $p_{2i-1}$ of $F$.

• If $x$ is divisible by $a_i$, and $i$ is the least integer, $1 \leq i \leq n$, for which this is true, then $\text{RULE}(F,x) = i$.

If $x$ is not divisible by any $a_i$, $1 \leq i \leq n$, then $x$ is divisible by 1, and $\text{RULE}(F,x)$ returns $n+1$. That’s why we added $p_{2n+1} \ p_{2n+2}$.

• Given the function $\text{RULE}(F,x)$, we can determine $\text{NEXT}(F,x)$, the number that follows $x$, when using $F$, by

$$\text{NEXT}(F, x) = (x \ // \ \exp(F, 2*\text{RULE}(F, x)-1)) \ * \ \exp(F, 2*\text{RULE}(F, x))$$
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 as our means of halting. Of course, no applicable rule meets this definition as the $n+1$-st rule divides and then multiplies the latest value by 1.
Simulation by Recursive # 3

• A Universal Machine that simulates an arbitrary Factor System, Turing Machine, Register Machine, Recursive Function can then be defined by

$$\text{Univ}(F, x) = \exp \left( \text{CONFIG} \left( F, x, \text{HALT} \left( F, x \right) \right), 0 \right)$$

• 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. It is that ability that makes this function universal.
FRS Subtraction

• $2^{03a5^b} \Rightarrow 2^{a-b}$
  $3*5x \rightarrow x$ or $1/15$
  $5x \rightarrow x$ or $1/5$
  $3x \rightarrow 2x$ or $2/3$

• Encode $F = 2^3 3^{15} 5^1 7^5 11^1 13^3 17^2 19^1 23^1$

• Consider $a=4$, $b=2$

• $RULE(F, x) = \mu z (1 \leq z \leq 4) \begin{bmatrix} \exp(F, 2*z-1) | x \end{bmatrix}$
  $RULE(F, 3^4 5^2) = 1$, as $15$ divides $3^4 5^2$

• $NEXT(F, x) = (x \div \exp(F, 2*RULE(F, x)-1)) \times \exp(F, 2*RULE(F, x))$
  $NEXT(F, 3^4 5^2) = (3^4 5^2 \div 15 \times 1) = 3^3 5^1$
  $NEXT(F, 3^3 5^1) = (3^3 5^1 \div 15 \times 1) = 3^2$
  $NEXT(F, 3^2) = (3^2 \div 3 \times 2) = 2^1 3^1$
  $NEXT(F, 2^1 3^1) = (2^1 3^1 \div 3 \times 2) = 2^2$
  $NEXT(F, 2^2) = (2^2 \div 1 \times 1) = 2^2$
Rest of simulation

- CONFIG(F, x, 0) = x
  CONFIG(F, x, y+1) = NEXT(F, CONFIG(F, x, y))

- CONFIG(F, 3^4 5^2, 0) = 3^4 5^2
  CONFIG(F, 3^4 5^2, 1) = 3^3 5^1
  CONFIG(F, 3^4 5^2, 2) = 3^2
  CONFIG(F, 3^4 5^2, 3) = 2^1 3^1
  CONFIG(F, 3^4 5^2, 4) = 2^2
  CONFIG(F, 3^4 5^2, 5) = 2^2

- HALT(F, x) = \mu y[CONFIG(F, x, y) == CONFIG(F, x, y+1)] = 4

- Univ(F, x) = \exp(2^2, 0) = 2
Simplicity of Universal

- A side result is that every computable (recursive) function can be expressed in the form

\[ F(x) = G(\mu y \ H(x, y)) \]

where \( G \) and \( H \) are primitive recursive.
RECURSIVE $\leq$ TURING
Standard Turing Computation

- Our notion of standard Turing computability of some $n$-ary function $F$ assumes that the machine starts with a tape containing the $n$ inputs, $x_1, \ldots, x_n$ in the form

  $$\ldots01^{x_1}01^{x_2}0\ldots01^{x_n}0\ldots$$

  and ends with

  $$\ldots01^{x_1}01^{x_2}0\ldots01^{x_n}01^y0\ldots$$

  where $y = F(x_1, \ldots, x_n)$. 
More Helpers

• To build our simulation we need to construct some useful submachines, in addition to the $R$, $L$, $R$, $L$, and $C_k$ machines already defined.

• $T$ -- translate moves a value left one tape square
  
  $\ldots\text{?}01^x0\ldots \Rightarrow \ldots\text{?}1^x00\ldots$

• Shift -- shift a rightmost value left, destroying value to its left
  
  $\ldots01^x101^x20\ldots \Rightarrow \ldots01^x20\ldots$

• $\text{Rot}_k$ -- Rotate a $k$ value sequence one slot to the left
  
  $\ldots01^x101^x20\ldots01^xk0\ldots$
  
  $\Rightarrow \ldots01^x20\ldots01^xk01^x10\ldots$
Basic Functions

All Basis Recursive Functions are Turing computable:

- \( C_a^n(x_1, \ldots, x_n) = a \)  
  \( (R1)^aR \)

- \( I_i^n(x_1, \ldots, x_n) = x_i \)  
  \( C_{n-i+1} \)

- \( S(x) = x+1 \)  
  \( C_11R \)
Closure Under Composition

If \( G, H_1, \ldots, H_k \) are already known to be Turing computable, then so is \( F \), where

\[
F(x_1, \ldots, x_n) = G(H_1(x_1, \ldots, x_n), \ldots, H_k(x_1, \ldots, x_n))
\]

To see this, we must first show that if \( E(x_1, \ldots, x_n) \) is Turing computable then so is

\[
E<m>(x_1, \ldots, x_n, y_1, \ldots, y_m) = E(x_1, \ldots, x_n)
\]

This can be computed by the machine

\[
\mathcal{L}^{n+m} (\text{Rot}_{n+m})^n \mathcal{R}^{n+m} E \mathcal{L}^{n+m+1} (\text{Rot}_{n+m})^m \mathcal{R}^{n+m+1}
\]

Can now define \( F \) by

\[
H_1 \text{ } H_2<1> \text{ } H_3<2> \ldots \text{ } H_k<k-1> \text{ } G \text{ } \text{Shift}^k
\]
Closure Under Induction

To prove that Turing Machines are closed under induction (primitive recursion), we must simulate some arbitrary primitive recursive function $F(y, x_1, x_2, \ldots, x_n)$ on a Turing Machine, where

$F(0, x_1, x_2, \ldots, x_n) = G(x_1, x_2, \ldots, x_n)$

$F(y+1, x_1, x_2, \ldots, x_n) = H(y, x_1, x_2, \ldots, x_n, F(y, x_1, x_2, \ldots, x_n))$

Where, $G$ and $H$ are Standard Turing Computable. We define the function $F$ for the Turing Machine as follows:

Since our Turing Machine simulator can produce the same value for any arbitrary PRF, $F$, we show that Turing Machines are closed under induction (primitive recursion).
Closure Under Minimization

If $G$ is already known to be Turing computable, then so is $F$, where

$$F(x_1, \ldots, x_n) = \mu y (G(x_1, \ldots, x_n, y) == 1)$$

This can be done by

$$\begin{array}{cccccc}
  & R & G & L & 0 & L \\
 0 & \ 1 & \ 0 & 1 & \ \\
 1 & \ 1 & \ \\
\end{array}$$
Consequences of Equivalence

• Theorem: The computational power of Recursive Functions, Turing Machines, Register Machine, and Factor Replacement Systems are all equivalent.

• Theorem: Every Recursive Function (Turing Computable Function, etc.) can be performed with just one unbounded type of iteration.

• Theorem: Universal machines can be constructed for each of our formal models of computation.
Additional Notations

Includes comment on our notation versus that of others
Universal Machine

• Others consider functions of $n$ arguments, whereas we had just one. However, our input to the FRS was actually an encoding of $n$ arguments.

• The fact that we can focus on just a single number that is the encoding of $n$ arguments is easy to justify based on the pairing function.

• Some presentations order arguments differently, starting with the $n$ arguments and then the Gödel number of the function, but closure under argument permutation follows from closure under substitution.
Universal Machine Mapping

- $\varphi^{(n)}(f, x_1, \ldots, x_n) = \text{Univ}(f, \prod_{i=1}^{n} p_{i}^{x_i})$

- We will sometimes adopt the above and also its common shorthand

  $\varphi_f^{(n)}(x_1, \ldots, x_n) = \varphi^{(n)}(f, x_1, \ldots, x_n)$

  and the even shorter version

  $\varphi_f(x_1, \ldots, x_n) = \varphi^{(n)}(f, x_1, \ldots, x_n)$
SNAP and TERM

• Our `CONFIG` is essentially a snapshot function as seen in other presentations of a universal function

\[
\text{SNAP}(f, x, t) = \text{CONFIG}(f, x, t)
\]

• Termination in our notation occurs when we reach a fixed point, so

\[
\text{TERM}(f, x) = (\text{NEXT}(f, x) == x)
\]

• Again, we used a single argument but that can be extended as we have already shown.
STP Predicate

• **STP**\((f, x_1,\ldots,x_n, t)\) is a predicate defined to be true iff \(\phi_f (x_1,\ldots,x_n)\) converges in at most \(t\) steps.

• **STP** is primitive recursive since it can be defined by

\[
\text{STP}(f, x, s) = \text{TERM}(f, \text{CONFIG}(f, x, s))
\]

Extending to many arguments is easily done as before.
VALUE PRF

• \( \text{VALUE}(f, x_1, \ldots, x_n, t) \) is a primitive recursive function (algorithm) that returns \( \varphi_f(x_1, \ldots, x_n) \) so long as \( \text{STP}(f, x_1, \ldots, x_n, t) \) is true.

• \( \text{VALUE}(f, x_1, \ldots, x_n, t) \) returns a value if \( \text{STP}(f, x_1, \ldots, x_n, t) \) is false, but the returned value is meaningless.
Recursively Enumerable

Properties of re Sets
Definition of re

• Some texts define re in the same way as I have defined semi-decidable.
  \( S \subseteq \mathbb{N} \) is semi-decidable iff there exists a partially computable function \( g \) where
  \[
  S = \{ x \in \mathbb{N} \mid g(x)\downarrow \}
  \]

• I prefer the definition of re that says
  \( S \subseteq \mathbb{N} \) is re iff \( S = \emptyset \) or there exists a totally computable function \( f \) where
  \[
  S = \{ y \mid \exists x \ f(x) == y \}
  \]

• We will prove these equivalent. Actually, \( f \) can be a primitive recursive function.
Semi-Decidable Implies re

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

$$F_{\mathcal{S}}(<x,t>) = x \ast \text{STP}(g_\mathcal{S}, x, t) + a \ast (1 - \text{STP}(g_\mathcal{S}, x, t))$$

Note: $F_\mathcal{S}$ is primitive recursive and it enumerates every value in $\mathcal{S}$ 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

$$\mu y \ [y == y+1] \text{ if } S == \emptyset$$

$$\psi_S(x) = \text{signum}((\mu y[F_S(y)==x])+1), \text{ otherwise}$$

This achieves our result as the domain of $\psi_S$ is the range of $F_S$, or empty if $S == \emptyset$. Note that this is an existence proof in that we cannot test if $S == \emptyset$. 

2/8/19

© UCF EECS 360
Domain of a Procedure

Corollary: $S$ is re/semi-decidable iff $S$ is the domain / range of a partial recursive predicate $F_S$.

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

$$\mu y \ [y == y+1] \quad \text{if } S == \emptyset$$

$$\psi_S(x) =$$

$$x*\text{signum}((\mu y[F_S(y)==x])+1), \text{ otherwise}$$
Recursive Implies re

Theorem: Recursive implies re.
Proof: $S$ is recursive implies there is a total recursive function $f_S$ such that

$$S = \{ x \in \mathbb{N} \mid f_S(x) == 1 \}$$

Define $g_S(x) = \mu y \ (f_S(x) == 1)$

Clearly

$$\text{dom}(g_S) = \{ x \in \mathbb{N} \mid g_S(x) \downarrow \}$$

$$= \{ x \in \mathbb{N} \mid f_S(x) == 1 \}$$

$$= S$$
Related Results

Theorem: $S$ is re iff $S$ is semi-decidable.
Proof: That’s what we proved.

Theorem: $S$ and $\neg S$ are both re (semi-decidable) iff $S$ (equivalently $\neg S$) is recursive (decidable).
Proof: Let $f_S$ semi-decide $S$ and $f_S'$ semi-decide $\neg S$. We can decide $S$ by $g_S$

$$g_S(x) = \text{STP}(f_S, x, \mu t \ (\text{STP}(f_S, x, t) \ || \ \text{STP}(f_S', x, t)))$$

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

The other direction is immediate since, if $S$ is decidable then $\neg S$ is decidable (just complement $g_S$) and hence they are both re (semi-decidable).
Enumeration Theorem

• Define
  \[ W_n = \{ x \in \mathbb{N} \mid \varphi(n,x) \downarrow \} \]
  
• Theorem: A set \( B \) is re iff there exists an \( n \) such that \( B = W_n \).
  
Proof: Follows from definition of \( \varphi(n,x) \).

• This gives us a way to enumerate the recursively enumerable sets.

• Note: We will later show (again) that we cannot enumerate the recursive sets.
The Set $K$

- $K = \{ n \in \mathbb{N} \mid n \in W_n \}$
- Note that $n \in W_n \iff \varphi(n,n) \downarrow \iff \text{HALT}(n,n)$
- Thus, $K$ is the set consisting of the indices of each program that halts when given its own index
- $K$ can be semi-decided by the HALT predicate above, so it is re.
K is not Recursive

• Theorem: We can prove this by showing \( \neg K \) is not re.

• If \( \neg K \) is re then \( \neg K = W_i \), for some \( i \).

• However, this is a contradiction since
  \[
  i \in K \iff i \in W_i \iff i \in \neg K \iff i \notin K
  \]
re Characterizations

Theorem: If \( S \neq \emptyset \) 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 recursive function
4. \( S \) is the range of a partial rec. function
5. \( S \) is the domain of a partial rec. function
6. \( S \) 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. Below, assume \( f_S \) enumerates \( S \).

\[
g_S(x) = x^*\text{STP}(f_S, x, \mu t (\text{STP}(f_S, x, t))) \text{ or } g_S(x) = x^* \exists t \text{STP}(f_S, x, t)
\]
S-m-n Theorem
Parameter (S-m-n) Theorem

• Theorem: For each $n, m > 0$, there is a prf $S_m^n(y, u_1, ..., u_n)$ such that

$$
\varphi^{(m+n)}(y, x_1, ..., x_m, u_1, ..., u_n) = \varphi^{(m)}(S_m^n(y, u_1, ..., u_n), x_1, ..., x_m)
$$

• The proof of this is highly dependent on the system in which you proved universality and the encoding you chose.
S-m-n for FRS

- We would need to create a new FRS, from an existing one \( F \), that fixes the value of \( u_i \) as the exponent of the prime \( p_{m+i} \).
- Sketch of proof:
  Assume we normally start with \( p_1^{x_1} \cdots p_m^{x_m} p_1^{u_1} \cdots p_{m+n}^{u_n} \sigma \)
  Here the first \( m \) are variable; the next \( n \) are fixed; \( \sigma \) denotes prime factors used to trigger first phase of computation.
  Assume that we use fixed point as convergence.
  We start with just \( p_1^{x_1} \cdots p_m^{x_m} \), with \( q \) the first unused prime.

\[
q \alpha x \rightarrow q \beta x \quad \text{replaces } \alpha x \rightarrow \beta x \text{ in } F, \text{ for each rule in } F
\]
\[
q x \rightarrow q x \quad \text{ensures we loop at end}
\]
\[
x \rightarrow q p_{m+1}^{u_1} \cdots p_{m+n}^{u_n} \sigma x \quad \text{adds fixed input, start state and } q
\]
  this is selected once and never again

Note: \( q = \text{prime}(\max(n+m, \text{lastFactor}(\text{Product}[i=1 \text{ to } r] \alpha_i \beta_i ))+1) \)
  where \( r \) is the number of rules in \( F \).
Details of S-m-n for FRS

• The number of F (called F, also) is $2^r 3^{a_1} 5^{b_1} \cdots p_{2r-1}^{a_r} p_{2r}^{b_r}$

• $S_{m,n}(F, u_1, \ldots, u_n) = 2^{r+2} q^{x_1} 5^{q x_1} \cdots p_{2r-1}^{q x_{a_r}} p_{2r}^{q x_{b_r}}\ p_{2r+1}^q p_{2r+2}^q p_{2r+3} p_{2r+4} q p_{m+1} u_1 \ldots p_{m+n} u_n \sigma$

• This represents the rules we just talked about. The first added rule pair means that if the algorithm does not use fixed point, we force it to do so. The last rule pair is the only one initially enabled and it adds the prime $q$, the fixed arguments $u_1, \ldots, u_n$, the enabling prime $q$, and the $\sigma$ needed to kick start computation. Note that $\sigma$ could be a 1, if no kick start is required.

• $S_{m,n} = S_m^n$ is clearly primitive recursive. I’ll leave the precise proof of that as a challenge to you.
Quantification 1 & 2
Quantification#1

- \( S \) is decidable iff there exists an algorithm \( \chi_S \) (called \( S \)'s characteristic function) such that
  \[ x \in S \iff \chi_S(x) \]
  This is just the definition of decidable.

- \( S \) is re iff there exists an algorithm \( A_S \) where
  \[ x \in S \iff \exists t \ A_S(x,t) \]
  This is clear since, if \( g_S \) is the index of the procedure \( \psi_S \) that semi-decides \( S \) then
  \[ x \in S \iff \exists t \ \text{STP}(g_S, x, t) \]
  So, \( A_S(x,t) = \text{STP}_{g_S}(x, t) \), where \( \text{STP}_{g_S} \) is the \( \text{STP} \) function with its first argument fixed.

- Creating new functions by setting some one or more arguments to constants is an application of \( S_{m^n} \).
Quantification#2

• S is re iff there exists an algorithm $A_S$ such that $x \notin S \iff \forall t \ A_S(x,t)$
  This is clear since, if $g_S$ is the index of the procedure $\psi_S$ that semi-decides $S$, then
  $x \notin S \iff \exists t \ \text{STP}(g_S, x, t) \iff \forall t \ \neg \text{STP}(g_S, x, t)$
  So, $A_S(x,t) = \neg \text{STP}_{g_S}(x, t)$, where $\text{STP}_{g_S}$ is the \text{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.
Diagonalization and Reducibility
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 algorithms can be enumerated, and that \( F \) accomplishes this. Then

\[
F(x) = F_x
\]

where \( F_0, F_1, F_2, \ldots \) is a list of all the algorithms
The Contradiction

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

- 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 TOT

• The listing of all algorithms can be viewed as

\[ \text{TOT} = \{ f \in \mathbb{N} \mid \forall x \ \varphi(f, x) \downarrow \} \]

• We can also note that

\[ \text{TOT} = \{ f \in \mathbb{N} \mid W_f = \mathbb{N} \} \]

• Theorem: TOT is not re.
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.
Diagonalization is a Bummer

• The issues with diagonalization are that it is tedious and is applicable as a proof of undecidability or non-re-ness for only a small subset of the problems that interest us.

• Thus, we will now seek to use reduction wherever possible.

• To show a set, $S$, is undecidable, we can show it is as least as hard as the set $K_0$. That is, $K_0 \leq S$. Here the mapping used in the reduction does not need to run in polynomial time, it just needs to be an algorithm.

• To show a set, $S$, is not re, we can show it is as least as hard as the set TOTAL (the set of algorithms). That is, $TOTAL \leq S$. 
Reduction to TOTAL

• We can show that the set $K_0$ (Halting) is no harder than the set $\text{TOTAL}$ (Uniform Halting). Since we already know that $K_0$ is unsolvable, we would now know that $\text{TOTAL}$ is also unsolvable. We cannot reduce in the other direction since $\text{TOTAL}$ is in fact harder than $K_0$.

• Let $\phi_F$ be some arbitrary effective procedure and let $x$ be some arbitrary natural number.

• Define $F_x(y) = \phi_F(x)$, for all $y \in \mathbb{N}$

• Then $F_x$ is an algorithm if and only if $\phi_F$ halts on $x$.

• Thus, $K_0 \leq \text{TOTAL}$, and so a solution to membership in $\text{TOTAL}$ would provide a solution to $K_0$, which we know is not possible.
Reduction to ZERO

• We can show that the set TOTAL is no harder than the set-zero = \{ f | \forall x \ \varphi_f(x) = 0 \}. Since we already know that TOTAL is non-re, we would now know that ZERO is also non-re.

• Let \varphi_f be some arbitrary effective procedure.

• Define \( F_f(y) = \varphi_f(x) - \varphi_f(x), \) for all \( x \in \mathbb{N} \)

• Then \( F_f \) is an algorithm that produces 0 for all input (is in the set ZERO) if and only if \( \varphi_f \) halts on all input \( x \). Thus, \( TOTAL \leq ZERO \).

• Thus a semi-decision procedure for ZERO would provide one for TOTAL, a set already known to be non-re.
Classic Undecidable Sets

• The universal language
  \[ K_0 = L_u = \{ <f, x> | \varphi_f(x) \text{ is defined} \} \]

• Membership problem for \( L_u \) is the **Halting Problem**.

• The sets \( L_{ne} \) and \( L_e \), where

  \[
  \text{NON-EMPTY} = L_{ne} = \{ f | \exists x \varphi_f(x) \downarrow \} \\
  \text{EMPTY} = L_e = \{ f | \forall x \varphi_f(x) \uparrow \}
  \]

  are the next ones we will study.
$L_{ne}$ is re

- $L_{ne}$ is enumerated by
  
  \[ F(<f, x, t>) = f \times \text{STP}(f, x, t) \]

- This assumes that 0 is in $L_{ne}$ since 0 probably encodes some trivial machine. If this isn’t so, we’ll just slightly vary our enumeration of the recursive functions so it is true.

- Thus, the range of this total function $F$ is exactly the indices of functions that converge for some input, and that’s $L_{ne}$. 
L_{ne} is Non-Recursive

• Note in the previous enumeration that F is a function of just one argument, as we are using an extended pairing function \(<x,y,z> = <x,\langle y,z \rangle>\).
• Now L_{ne} cannot be recursive, for if it were then L_{u} (K_{0}) is recursive by the reduction we showed before.
• In particular, from any index \(x\) and input \(y\), we created a new function which accepts all input just in case the \(x\)-th function accepts \(y\). Recall \(F_{x}(y) = \varphi_{F}(x)\), for all \(y \in \mathbb{N}\).
• Hence, this new function’s index is in L_{ne} just in case \(<x, y>\) is in L_{u} (K_{0}).
• Thus, a decision procedure for L_{ne} (equivalently for L_{e}) implies one for L_{u} (K_{0}).
L_{ne} is re by Quantification

• Can do by observing that

\[ f \in L_{ne} \iff \exists <x,t> \ STP(f, x, t) \]

• By our earlier results, any set whose membership can be described by an existentially quantified recursive predicate is re (semi-decidable).
Le is not re

• If Le were re, then L_{ne} would be recursive since it and its complement would be re.
• Can also observe that Le is the complement of an re set since

\[ f \in L_e \iff \forall \langle x, t \rangle \sim \text{STP}(f, x, t) \]
\[ \iff \sim \exists \langle x, t \rangle \text{STP}(f, x, t) \]
\[ \iff f \notin L_{ne} \]
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 a total recursive function $f$ such that $x \in A \iff f(x) \in B$
• 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: $\emptyset$, $\mathbb{N}$, 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 total recursive 1-1 function $f$ such that
  \[ x \in A \iff f(x) \in B \]
• 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

- A 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 $\text{HALT}$ 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 $\text{Halt} = K_0 = L_u$

- $\text{Halt} = K_0 = L_u = \{ <f, x> \mid \varphi_f(x) \downarrow \}$
- Let $A$ be an arbitrary re set. By definition, there exists an effective procedure $\varphi_a$, 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 $<a, x> \in K_0$
- The above provides a 1-1 function that reduces $A$ to $K_0$ ($A \leq_1 K_0$)
- Thus the universal set, $\text{Halt} = K_0 = L_u$, is an re (1-1, m-1, Turing) complete set.
The Set $K$

- $K = \{ f \mid \varphi_f(f) \text{ is defined} \}$

- Define $f_x(y) = \varphi_f(x)$, for all $y$. The index for $f_x$ can be computed from $f$ and $x$ using $S_{1,1}$, where we add a dummy argument, $y$, to $\varphi_f$. Let that index be $f_x$. (Yeah, that’s overloading.)

- $\langle f, x \rangle \in K_0$ iff $x \in \text{dom}(\varphi_f)$ iff $\forall y[\varphi_{f_x}(y) \downarrow]$ iff $f_x \in K$.

- 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.
Quantification # 3 and the Overall Picture
The **Uniform Halting Problem** was already shown to be non-re. It turns out its complement is also not re. We’ll cover that later. In fact, we will show that **TOT** requires an alternation of quantifiers. Specifically,

\[ f \in \text{TOT} \iff \forall x \exists t \ ( \text{STP}(f, x, t) ) \]

and this is the minimum quantification we can use, given that the quantified predicate is total recursive (actually primitive recursive here).
NonRE = (NRNC ∪ Co-RE) - REC
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, \textit{wlog}, that $P$ does not contain $\emptyset$. If it does we switch our attention to the complement of $P$. Now, since $P$ is non-trivial, there exists some language $L$ with property $P$. Let $[r]$ be a recursive function whose domain is $L$ ($r$ is the index of a semi-decision procedure for $L$). Suppose $P$ were decidable. We will use this decision procedure and the existence of $r$ to decide $K_0$. 
First we define a function $F_{r,x,y}$ for $r$ and each function $\phi_x$ and input $y$ as follows.

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

The domain of this function is $L$ if $\phi_x(y)$ 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 $\phi_x(y)$ 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 regular 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 Property

• 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, $f$ and $g$, are such that $\varphi_f$ and $\varphi_g$ converge on the same inputs and, when they converge, produce precisely the same result, then both $f$ and $g$ 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

$$S_\mathcal{P} = \{ x \mid \varphi_x \text{ has property } \mathcal{P} \}$$

is undecidable. Note that membership in $S_\mathcal{P}$ is based purely on the input/output behavior of a function, not on any aspect of its implementation.
Strong Rice’s Proof

• Given \( x, y, r \), where \( r \) is in the set \( S_P = \{ f \mid \varphi_f \text{ has property } P \} \), define the function
  \[
  f_{x,y,r}(z) = \varphi_x(y) - \varphi_x(y) + \varphi_r(z).
  \]

• \( f_{x,y,r}(z) = \varphi_r(z) \) if \( \varphi_x(y) \downarrow ; = \emptyset \) if \( \varphi_x(y) \uparrow \).
  Thus, \( \varphi_x(y) \downarrow \) iff \( f_{x,y,r} \) has property \( P \), and so \( K_0 \leq S_P \).
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
Corollaries to Rice’s

Corollary: The following properties of regular sets are undecidable

a) \( L = \emptyset \)

b) \( L \) is finite

c) \( L \) is a regular set

d) \( L \) is a context-free set
Practice

Known Results:

HALT = \{ <f,x> \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 NonTrivial, where
   NonTrivial = \{ f \mid \text{for some } x, y, x \neq y, f(x) \downarrow \text{ and } f(y) \downarrow \text{ and } f(x) \neq f(y) \} 

2. Show that Non-Trivial reduces to HALT. (1 plus 2 show they are equally hard)

3. Use Reduction from TOTAL to show that NoRepeats is not even re, where
   NoRepeats = \{ f \mid \text{for all } x, y, f(x) \downarrow \text{ and } f(y) \downarrow, \text{ and } x \neq y \Rightarrow f(x) \neq f(y) \} 

4. Show NoRepeats reduces to TOTAL. (3 plus 4 show they are equally hard)

5. Use Rice’s Theorem to show that NonTrivial is undecidable

6. Use Rice’s Theorem to show that NoRepeats is undecidable
Practice Classifications

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) \( \text{NonTrivial} = \{ f \mid \text{for some } x, y, x \neq y, f(x)\downarrow \text{ and } f(y)\downarrow \text{ and } f(x) \neq f(y) \} \)
   b) \( \text{NoRepeats} = \{ f \mid \text{for all } x, y, f(x)\downarrow \text{ and } f(y)\downarrow, \text{ and } x \neq y \Rightarrow f(x) \neq f(y) \} \)
   c) \( \text{FIN} = \{ f \mid \text{domain}(f) \text{ is finite} \} \)

2. Let set \( A \) be non-empty recursive, and let \( B \) be re non-recursive. Consider \( C = \{ z \mid z = x \ast y, \text{where } x \in A \text{ and } y \in B \} \). 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?
Sample Question#1

1. Given that the predicate $\text{STP}$ and the function $\text{VALUE}$ are algorithms, show that we can semi-decide

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

Note: $\text{STP}(f, x, s)$ is true iff $\varphi_f(x)$ converges in $s$ or fewer steps and, if so, $\text{VALUE}(f, x, s) = \varphi_f(x)$. 
Sample Questions #2, 3

2. Use Rice’s Theorem to show that $\text{HZ}$ is undecidable, where $\text{HZ}$ is

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

3. Redo using Reduction from $\text{HALT}$.
4. Let $P = \{ f \mid \exists x \ [ \text{STP}(f, x, x) ] \}$. Why does Rice’s theorem not tell us anything about the undecidability of $P$?
5. Let $S$ be an re (recursively enumerable), non-recursive set, and $T$ be an re, possibly recursive non-empty set. Let $E = \{ z \mid z = x + y, \text{ where } x \in S \text{ and } y \in T \}$. 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?
Constant time:
Not amenable to Rice’s
Constant Time

- $\text{CTime} = \{ M \mid \exists K \ [ M \text{ halts in at most } K \text{ steps independent of its starting configuration } ] \}$
- $\text{RT}$ cannot be shown undecidable by Rice’s Theorem as it breaks property 2
  - Choose $M_1$ and $M_2$ to each Standard Turing Compute (STC) ZERO
  - $M_1$ is $R$ (move right to end on a zero)
  - $M_2$ is $L \ R \ R$ (time is dependent on argument)
  - $M_1$ is in $\text{CTime}$; $M_2$ is not, but they have same I/O behavior, so $\text{CTime}$ does not adhere to property 2
Quantifier Analysis

- \( \text{CTime} = \{ M \mid \exists K \forall C [ \text{STP}(M, C, K) ] \} \)

- This would appear to imply that \( \text{CTime} \) is not even re. However, a TM that only runs for \( K \) steps can only scan at most \( K \) distinct tape symbols. Thus, if we use unary notation, \( \text{CTime} \) can be expressed

- \( \text{CTime} = \{ M \mid \exists K \forall C_{|C| \leq K} [ \text{STP}(M, C, K) ] \} \)

- We can dovetail over the set of all TMs, \( M \), and all \( K \), listing those \( M \) that halt in constant time.
Complexity of CTime

• Can show it is equivalent to the **Halting Problem** for TM’s with **Infinite Tapes** (not unbounded but truly infinite)

• This was shown in 1966 to be undecidable.

• It was also shown to be re, just as we have done so for **CTime**.

• Details Later
Post Systems
Thue Systems

- Devised by Axel Thue
- Just a string rewriting view of finitely presented monoids
- $T = (\Sigma, R)$, where $\Sigma$ is a finite alphabet and $R$ is a finite set of bi-directional rules of form $\alpha_i \leftrightarrow \beta_i$, $\alpha_i, \beta_i \in \Sigma^*$
- We define $\Leftrightarrow^*$ as the reflexive, transitive closure of $\Leftrightarrow$, where $w \Leftrightarrow x$ iff $w = y\alpha z$ and $x = y\beta z$, where $\alpha \leftrightarrow \beta$
Semi-Thue Systems

- Devised by Emil Post
- A one-directional version of Thue systems
- \( S = (\Sigma, R) \), where \( \Sigma \) 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 \( \alpha \rightarrow \beta \)
Word Problems

• Let $S = (\Sigma, R)$ be some Thue (Semi-Thue) system, then the word problem for $S$ is the problem to determine of arbitrary words $w$ and $x$ over $S$, whether or not $w \iff^* x (w \implies^* x)$.

• The Thue system word problem is the problem of determining membership in equivalence classes. This is not true for Semi-Thue systems.

• We can always consider just the relation $\implies^*$ since the symmetric property of $\iff^*$ comes directly from the rules of Thue systems.
Post Canonical Systems

• These are a generalization of Semi-Thue systems.
• \( P = (\Sigma, V, R) \), where \( \Sigma \) is a finite alphabet, \( V \) is a finite set of “variables”, and \( R \) is a finite set of rules.
• Here the premise part (left side) of a rule can have many premise forms, e.g., a rule appears as
  \[ P_{1,1} \alpha_{1,1} P_{1,2} \cdots \alpha_{1,n_1} P_{1,n_1} \alpha_{1,n_1+1}, \]
  \[ P_{2,1} \alpha_{2,1} P_{2,2} \cdots \alpha_{2,n_2} P_{2,n_2} \alpha_{2,n_2+1}, \]
  \[ \ldots \]
  \[ P_{k,1} \alpha_{k,1} P_{k,2} \cdots \alpha_{k,n_k} P_{k,n_k} \alpha_{k,n_k+1}, \]
  \[ \rightarrow Q_1 \beta_1 Q_2 \cdots \beta_{n_k+1} Q_{n_k+1} \beta_{n_k+1+1} \]
• In the above, the \( P \)’s and \( Q \)’s are variables, the \( \alpha \)’s and \( \beta \)’s are strings over \( \Sigma \), and each \( Q \) must appear in at least one premise.
• We can extend the notion of \( \Rightarrow^* \) to these systems considering sets of words that derive conclusions. Think of the original set as axioms, the rules as inferences and the final word as a theorem to be proved.
Examples of Canonical Forms

• Propositional rules
  \( P, P \supset Q \rightarrow Q \)
  \( \neg P, P \cup Q \rightarrow Q \)
  \( P \cap Q \rightarrow P \)
  \( P \cap Q \rightarrow Q \)
  \( (P \cap Q) \cap R \leftrightarrow P \cap (Q \cap R) \)
  \( (P \cup Q) \cup R \leftrightarrow P \cup (Q \cup R) \)
  \( \neg (\neg P) \leftrightarrow P \)
  \( P \cup Q \rightarrow Q \cup P \)
  \( P \cap Q \rightarrow Q \cap P \)

• Some proofs over \( \{a,b,(),\neg,\supset,\cup,\cap\} \)
  \( \{a \cup c, b \supset \neg c, b\} \Rightarrow \{a \cup c, b \supset \neg c, b, \neg c\} \Rightarrow \)
  \( \{a \cup c, b \supset \neg c, b, \neg c, c \cup a\} \Rightarrow \)
  \( \{a \cup c, b \supset \neg c, b, \neg c, c \cup a, a\} \) which proves “a”
Simplified Canonical Forms

• Each rule of a Semi-Thue system is a canonical rule of the form
  \[ P\alpha Q \rightarrow P\beta Q \]
• Each rule of a Thue system is a canonical rule of the form
  \[ P\alpha Q \leftrightarrow P\beta Q \]
• Each rule of a Post Normal system is a canonical rule of the form
  \[ \alpha P \rightarrow P\beta \]
• Tag systems are just Normal systems where all premises are of the same length (the deletion number), and at most one can begin with any given letter in \( \Sigma \). That makes Tag systems deterministic.
Examples of Post Systems

• Alphabet $\Sigma = \{a, b, \#\}$. Semi-Thue rules:
  $aba \rightarrow b$
  $\#b\# \rightarrow \lambda$
  For above, $#a^nba^m# \Rightarrow^* \lambda$ iff $n=m$

• Alphabet $\Sigma = \{0, 1, c, \#\}$. Normal rules:
  $0c \rightarrow 1$
  $1c \rightarrow c0$
  $\#c \rightarrow \#1$
  $0 \rightarrow 0$
  $1 \rightarrow 1$
  $\# \rightarrow \#$
  For above, $binaryc\# \Rightarrow^* binary+1\#$ where $binary$ is some binary number.
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 $h\alpha qa\beta h$, $a$ being the scanned symbol.
• The tuple $qa b s$ leads to $qa \rightarrow sb$
• Moving right and left can be harder due to blanks.
Details of Halt(TM) ≤ Word(ST)

Let M = (Q, {0,1}, T), T is Turing table.

If qabs ∈ T, add rule qa → sb

If qaRs ∈ T, add rules

- q1b → 1sb a=1, ∀b ∈ {0,1}
- q1h → 1s0h a=1
- cq0b → c0sb a=0, ∀b,c ∈ {0,1}
- hq0b → hsb a=0, ∀b ∈ {0,1}
- cq0h → c0s0h a=0, ∀c ∈ {0,1}
- hq0h → hs0h a=0

If qaLs ∈ T, add rules

- bqac → sbac a,b,c ∈ {0,1}
- hqac → hs0ac a,c ∈ {0,1}
- bq1h → sb1h a=1, ∀b ∈ {0,1}
- hq1h → hs01h a=1
- bq0h → sbh a=0, ∀b ∈ {0,1}
- hq0h → hs0h a=0
Clean-Up

- Assume $q_1$ is start state and only one accepting state exists $q_0$
- We will start in $h1^xq_10h$, seeking to accept $x$ (enter $q_0$) or reject (run forever).
- Add rules
  - $q_0a \rightarrow q_0 \quad \forall a \in \{0,1\}$
  - $bq_0 \rightarrow q_0 \quad \forall b \in \{0,1\}$

- The added rule allows us to “erase” the tape if we accept $x$.
- This means that acceptance can be changed to generating $hq_0h$.

- The next slide shows the consequences.
Semi-Thue Word Problem

• Construction from TM, M, gets:
  • $h_1^xq_10h \Rightarrow_{\Sigma(M)}^* hq_0h$ iff $x \in \mathcal{L}(M)$.
  • $hq_0h \Rightarrow_{\Pi(M)}^* h_1^xq_10h$ iff $x \in \mathcal{L}(M)$.
  • $hq_0h \Leftrightarrow_{\Sigma(M)}^* h_1^xq_10h$ iff $x \in \mathcal{L}(M)$.

• Can recast both Semi-Thue and Thue Systems to ones over alphabet \{a,b\} or \{0,1\}. That is, a binary alphabet is sufficient for undecidability.
Formal Language

Undecidability Continued
PCP and Traces
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 $n>0$, $\Sigma$ a finite alphabet, and two $n$-tuples of words $(x_1, \ldots, x_n)$, $(y_1, \ldots, y_n)$ over $\Sigma$, does there exist a sequence $i_1, \ldots, i_k$, $k>0$, $1 \leq i_j \leq n$, such that $x_{i_1} \ldots x_{i_k} = y_{i_1} \ldots y_{i_k}$?

• Example of PCP:
  $n = 3$, $\Sigma = \{ a, b \}$, $(a b a, b b, a)$, $(b a b, b, b a a)$.
  Solution 2, 3, 1, 2
  $b b, a, a b a, b b = b b a a, b a b, b$
PCP Example#2

- Start with Semi-Thue System
  - \(aba \rightarrow ab; a \rightarrow aa; b \rightarrow a\)
  - Instance of word problem: \(bbbb \Rightarrow *? aa\)

- Convert to PCP
  - \([bbbb^* ab \quad ab \quad aa \quad aa \quad a \quad a \quad a ]\)
  - \([\quad aba \quad aba \quad a \quad a \quad b \quad b \quad *aa]\)
  - And \(\quad * \quad * \quad a \quad a \quad b \quad b\)
  - \(\quad * \quad * \quad a \quad a \quad b \quad b\)
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)
Ambiguity of CFG

• Problem to determine if an arbitrary CFG is ambiguous

\[ S \to A \mid B \]

\[ A \to x_i A \ [i] \mid x_i \ [i] \quad 1 \leq i \leq n \]

\[ B \to y_i B \ [i] \mid y_i \ [i] \quad 1 \leq i \leq n \]

\[ A \Rightarrow^* x_{i_1} \ldots x_{i_k} \ [i_k] \ldots \ [i_1] \quad k > 0 \]

\[ B \Rightarrow^* y_{i_1} \ldots y_{i_k} \ [i_k] \ldots \ [i_1] \quad k > 0 \]

• Ambiguous if and only if there is a solution to this PCP instance.
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.
CSG Produces Something

\[
S \rightarrow x_i S y_i^R | x_i T y_i^R \quad 1 \leq i \leq n
\]

\[
a T a \rightarrow * T *
\]

\[
* a \rightarrow a *
\]

\[
a * \rightarrow * a
\]

\[
T \rightarrow *
\]

• Our only terminal is *. We get strings of form \(*^{2j+1}\), for some j’s if and only if there is a solution to this PCP instance.
Review
1. Prove that the following are equivalent
   a) S is an infinite recursive (decidable) set.
   b) S is the range of a monotonically increasing total recursive function.
   Note: f is monotonically increasing means that $\forall x \ f(x+1) > f(x)$.
2. Let $A$ and $B$ be re sets. For each of the following, either prove that the set is re, or give a counterexample that results in some known non-re set.

a) $A \cup B$

b) $A \cap B$

c) $\sim A$
Sample Question#3

3. Present a demonstration that the $even$ function is primitive recursive.
   
   $\text{even}(x) = 1$ if $x$ is even
   
   $\text{even}(x) = 0$ if $x$ is odd
   
   You may assume only that the base functions are $\text{prf}$ and that $\text{prf}$’s are closed under a finite number of applications of composition and primitive recursion.
4. Given that the predicate \textbf{STP} and the function \textbf{VALUE} are prf’s, show that we can semi-decide

\{ f \mid \phi_f \text{ evaluates to 0 for some input} \}

Note: \textbf{STP} (f, x, s) is true iff \phi_f(x) converges in s or fewer steps and, if so, \textbf{VALUE}(f, x, s) = \phi_f(x).
5. Let $S$ be an re (recursively enumerable), non-recursive set, and $T$ be an re, possibly recursive set. Let $E = \{ z \mid z = x + y, \text{ where } x \in S \text{ and } y \in T \}$. 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?
6. Assuming that the Uniform Halting Problem (TOTAL) is undecidable (it’s actually not even re), use reduction to show the undecidability of

\[ \{ f \mid \forall x \ \phi_f(x+1) > \phi_f(x) \} \]
7. Let $\text{Incr} = \{ f | \forall x, \varphi_f(x+1) > \varphi_f(x) \}$. Let $\text{TOT} = \{ f | \forall x, \varphi_f(x) \downarrow \}$. Prove that $\text{Incr} \equiv_m \text{TOT}$. Note Q#6 starts this one.
8. Let $\text{Incr} = \{ f \mid \forall x \phi_f(x+1) > \phi_f(x) \}$. Use Rice’s theorem to show $\text{Incr}$ is not recursive.
Sample Question#9

9. Let $S$ be a recursive (decidable set), what can we say about the complexity (recursive, re, non-recursive, non-re) of $T$, where $T \subset S$?
10. Define the pairing function \( <x,y> \) and its two inverses \( <z>_1 \) and \( <z>_2 \), where if \( z = <x,y> \), then \( x = <z>_1 \) and \( y = <z>_2 \).
Sample Question#11

11. Assume $A \leq_m B$ and $B \leq_m C$. Prove $A \leq_m C$. 
Sample Question#12

12. Let $P = \{ f \mid \exists x \ [ \text{STP}(f, x, x) ] \}$. Why does Rice’s theorem not tell us anything about the undecidability of $P$?
What We've Done in Computability
List Minus Some Tedious Stuff

- A question with multiple parts that uses quantification (STP/VALUE)
- Various re and recursive equivalent definitions
- Proofs of equivalence of definitions
- Consequences of recursiveness or re-ness of a problem
- Closure of recursive/re sets
- Gödel numbering (pairing functions and inverses)
- Models of computation/equivalences (not details but understanding)
- Primitive recursion and its limitation; bounded versus unbounded μ
- Notion of universal machine
- A proof by diagonalization (there are just two possibilities)
- A question about K and/or K₀
- Many-one reduction(s)
- Rice’s Theorem (its proof and its variants)
- Applications of Rice’s Theorem and when it cannot be applied
Traces and Grammars
Traces (Valid Computations)

• A trace of a machine $M$, is a word of the form

$$\# X_0 \# X_1 \# X_2 \# X_3 \# \ldots \# X_{k-1} \# X_k \#$$

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 to then to intersect the two languages, so the result is a trace. This then allows us to create CFLs $L_1$ and $L_2$, where $L_1 \cap L_2 \neq \emptyset$, 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.
Traces of FRS

- I have chosen, once again to use the Factor Replacement Systems, but this time, Factor Systems with Residues. The rules are unordered and each is of the form $a \times x + b \rightarrow c \times x + d$

- These systems need to overcome the lack of ordering when simulating Register Machines. This is done by
  
  j. $\text{INC}_r[i]$: 
  $p_{n+j} x \rightarrow p_{n+i} \ p_r \ x$

  j. $\text{DEC}_r[s, f]$: 
  $p_{n+j} \ p_r \ x \rightarrow p_{n+s} \ x$
  $p_{n+j} \ p_r x + k \ p_{n+j} \rightarrow p_{n+f} \ p_r x + k \ p_{n+f}, 1 \leq k < p_r$

  We also add the halting rule associated with $m+1$ of
  $p_{n+m+1} x \rightarrow 0$

- Thus, halting is equivalent to producing 0. We can also add one more rule that guarantees we can reach 0 on both odd and even numbers of moves
  $0 \rightarrow 0$
Intersection of CFLs

- Let \((n, ((a_1,b_1,c_1,d_1), \ldots, (a_k,b_k,c_k,d_k))\) be some factor replacement system with residues. Define grammars \(G_1\) and \(G_2\) by using the \(4k+2\) rules

\[
G: \begin{align*}
F_i & \rightarrow 1^{a_i}F_i1^{c_i} | 1^{a_i+b_i}#1^{c_i+d_i} \quad 1 \leq i \leq k \\
S_1 & \rightarrow \# F_i S_1 | \# F_i \# \quad 1 \leq i \leq k \\
S_2 & \rightarrow \# 1^{x_0}S_11^{z_0}# \quad Z_0 \text{ is 0 for us}
\end{align*}
\]

- \(G_1\) starts with \(S_1\) and \(G_2\) with \(S_2\).

- Thus, using the notation of writing \(Y\) in place of \(1^Y\),

\[
L_1 = L( G_1 ) = \{ \# Y_0 \# Y_1 \# Y_2 \# Y_3 \# \ldots \# Y_{2j} \# Y_{2j+1} \# \}
\]

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

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

- But, \(L_2 = L( G_2 ) = \{ \# X_0 \# X_1 \# X_2 \# X_3 \# X_4 \# \ldots \# X_{2k-1} \# X_{2k} \# Z_0 \# \}
\]

where \(X_{2i-1} \Rightarrow X_{2i} , 1 \leq i \leq k\).

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

Now, $X_0$ is chosen as some selected input value to the Factor System with Residues, and $Z_0$ is the unique value (0 in our case) on which the machine halts. But,

$L_1 \cap L_2 = \{ \#X_0 \# X_1 \# X_2 \# X_3 \# X_4 \# \ldots \# X_{2k-1} \# X_{2k} \# Z_0 \# \}$

where $X_i \Rightarrow X_{i+1}$, $0 \leq i < 2k$, and $X_{2k} \Rightarrow Z_0$. This checks all steps of an even length computation. But our original system halts if and only if it produces 0 ($Z_0$) in an even (also odd) number of steps. Thus the intersection is non-empty just in case the Factor System with residue eventually produces 0 when started on $X_0$, just in case the Register Machine halts when started on the register contents encoded by $X_0$. 
Quotients of CFLs (concept)

Let \( L_1 = \{ \$ \# Y_0 \# Y_1 \# Y_2 \# Y_3 \# \ldots \# Y_{2j} \# Y_{2j+1} \# \} \) 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 \( L_2 = \{ X_0 \$ \# X_0 \# X_1 \# X_2 \# X_3 \# X_4 \# \ldots \# X_{2k-1} \# X_{2k} \# Z_0 \# \} \) where \( X_{2i-1} \Rightarrow X_{2i} \), \( 1 \leq i \leq k \) and \( Z \) 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 $.

Now, consider the quotient of \( L_2 / L_1 \). The only ways a member of \( L_1 \) can match a final substring in \( L_2 \) 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,
\[
L_2 / L_1 = \{ X_0 \mid \text{the system halts} \}.
\]

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.
Quotients of CFLs (precise)

• Let \((n, ((a_1,b_1,c_1,d_1), \ldots, (a_k,b_k,c_k,d_k))\) be some factor replacement system with residues. Define grammars \(G_1\) and \(G_2\) by using the \(4k+4\) rules:

\[
\begin{align*}
G &: F_i \rightarrow 1^{a_i}F_i1^{c_i} \mid 1^{a_i+b_i}\#1^{c_i+d_i} & 1 \leq i \leq k \\
T_1 & \rightarrow \#F_iT_1 \mid \#F_i\# & 1 \leq i \leq k \\
A & \rightarrow 1A1\mid \$\# \\
S_1 & \rightarrow \$T_1 \\
S_2 & \rightarrow A\ T_1\ \#1z_0\# & Z_0 \text{ is 0 for us}
\end{align*}
\]

\(G_1\) starts with \(S_1\) and \(G_2\) with \(S_2\).

• Thus, using the notation of writing \(Y\) in place of \(1^Y\),

\[
L_1 = L( G_1 ) = \{ \$\#Y_0\#Y_1\#Y_2\#Y_3\# \ldots \#Y_{2j}\#Y_{2j+1}\# \}
\]

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

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

But, \(L_2 = L( G_2 ) = \{ X\$\#X_0\#X_1\#X_2\#X_3\#X_4\# \ldots \#X_{2k-1}\#X_{2k}\#Z_0\# \}
\]

where \(X_{2i-1} \Rightarrow X_{2i}, 1 \leq i \leq k\) and \(X = X_0\).

This checks the odd/steps of an even length computation, and includes an extra copy of the starting number prior to its \$. 

2/8/19   © UCF EECS 463
Finish Quotient

Now, consider the quotient of $L_2 / L_1$. The only ways a member of $L_1$ can match a final substring in $L_2$ is to line up the $\$ \, \text{signs}$. 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, $L_2 / L_1 = \{ X \mid \text{the system F halts on zero} \}$. 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.
Traces and Type 0

- 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 $\Sigma$ that includes a blank symbol B.
- Consider a starting configuration C0. Our rules will be

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow # C0 #$</td>
<td>where $C0 = Yq0aX$ is initial ID</td>
</tr>
<tr>
<td>$q a \rightarrow s b$</td>
<td>if $q a b s \in T$</td>
</tr>
<tr>
<td>$b q a x \rightarrow b a s x$</td>
<td>if $q a R s \in T$, $a,b,x \in \Sigma$</td>
</tr>
<tr>
<td>$b q a # \rightarrow b a s B #$</td>
<td>if $q a R s \in T$, $a,b \in \Sigma$</td>
</tr>
<tr>
<td>$# q a x \rightarrow # a s x$</td>
<td>if $q a R s \in T$, $a,x \in \Sigma$, $a\neq B$</td>
</tr>
<tr>
<td>$# q a # \rightarrow # a s B #$</td>
<td>if $q a R s \in T$, $a \in \Sigma$, $a\neq B$</td>
</tr>
<tr>
<td>$# q a x \rightarrow # s x #$</td>
<td>if $q a R s \in T$, $x \in \Sigma$, $a=B$</td>
</tr>
<tr>
<td>$# q a # \rightarrow # s B #$</td>
<td>if $q a R s \in T$, $a=B$</td>
</tr>
<tr>
<td>$b q a x \rightarrow s b a x$</td>
<td>if $q a L s \in T$, $a,b,x \in \Sigma$</td>
</tr>
<tr>
<td>$# q a x \rightarrow # s B a x$</td>
<td>if $q a L s \in T$, $a,x \in \Sigma$</td>
</tr>
<tr>
<td>$b q a # \rightarrow s b a #$</td>
<td>if $q a L s \in T$, $a,b \in \Sigma$, $a\neq B$</td>
</tr>
<tr>
<td>$# q a # \rightarrow # s B a #$</td>
<td>if $q a L s \in T$, $a \in \Sigma$, $a\neq B$</td>
</tr>
<tr>
<td>$b q a # \rightarrow s b #$</td>
<td>if $q a L s \in T$, $b \in \Sigma$, $a=B$</td>
</tr>
<tr>
<td>$# q a # \rightarrow # s B #$</td>
<td>if $q a L s \in T$, $a=B$</td>
</tr>
<tr>
<td>$f \rightarrow \lambda$</td>
<td>if $f$ is a final state</td>
</tr>
<tr>
<td>$# \rightarrow \lambda$</td>
<td>just cleaning up the dirty linen</td>
</tr>
</tbody>
</table>

2/8/19 © UCF EECS 465
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 ∪ {S’, D}, T ∪ {d, e}, S’, P’), where P’ is

\[
\begin{align*}
S’ & \rightarrow eS e \\
D x & \rightarrow xD \quad \text{when } x \in V \cup T \\
D e & \rightarrow ed \quad \text{push the delete characters to far right} \\
\alpha & \rightarrow \beta \quad \text{where } \alpha \rightarrow \beta \in P \text{ and } |\alpha| \leq |\beta| \\
\alpha & \rightarrow \beta D^k \quad \text{where } \alpha \rightarrow \beta \in P \text{ and } |\alpha| - |\beta| = k > 0
\end{align*}
\]

Clearly, L(G’) = \{ e w e d^m | 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 $\lambda$-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.
Summary of Grammar Results
Decidability

- Everything about regular
- Membership in CFLs and CSLs
  - CKY for CFLs
- Emptiness for CFLs
Undecidability

• Is \( L = \emptyset \), for CSL, \( L \)?
• Is \( L = \Sigma^* \), for CFL (CSL), \( L \)?
• Is \( L_1 = L_2 \) for CFLs (CSLs), \( L_1, L_2 \)?
• Is \( L_1 \subseteq L_2 \) for CFLs (CSLs), \( L_1, L_2 \)?
• Is \( L_1 \cap L_2 = \emptyset \) for CFLs (CSLs), \( L_1, L_2 \)?
• Is \( L \) regular, for CFL (CSL), \( L \)?
• Is \( L_1 \cap L_2 \) a CFL for CFLs, \( L_1, L_2 \)?
• Is \( \neg L \) CFL, for CFL, \( L \)?
More Undecidability

• Is CFL, L, ambiguous?
• Is \( L = L^2 \), L a CFL?
• Does there exist a finite \( n \), \( L^n = L^{N+1} \)?
• Is \( L_1/L_2 \) finite, \( L_1 \) and \( L_2 \) CFLs?
• Membership in \( L_1/L_2 \), \( L_1 \) and \( L_2 \) CFLs?
Word to Grammar Problem

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

$G: S \rightarrow h1^xq_10h$
$hq_0h \rightarrow V$
$V \rightarrow aV$
$V \rightarrow \lambda$

• $x \in \mathcal{L}(M)$ 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 Grammar

• 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
  – $G2: S \rightarrow aS \mid \lambda \quad a \in \Sigma$
Turing Machine Traces

• A valid trace
  – $C_1 \# C_2^R \$ C_3 \# C_4^R \ldots \$ C_{2k-1} \# C_{2k}^R \$$, where $k \geq 1$ and $C_i \Rightarrow_M C_{i+1}$, for $1 \leq i < 2k$. Here, $\Rightarrow_M$ means derive in $M$, and $C^R$ means $C$ with its characters reversed.

• An invalid trace
  – $C_1 \# C_2^R \$ C_3 \# C_4^R \ldots \$ C_{2k-1} \# C_{2k}^R \$$, where $k \geq 1$ and for some $i$, it is false that $C_i \Rightarrow_M C_{i+1}$. 
What’s Context Free?

• Given a Turing Machine M
  – The set of invalid traces of M is Context Free
  – The set of valid traces is Context Sensitive
  – The set of valid terminating traces is Context Sensitive
  – The complement of the valid traces is Context Free
  – The complement of the valid terminating traces is Context Free
Partially correct traces

\[ L_1 = L( G_1 ) = \{ \#Y_0 \# Y_1 \# Y_2 \# Y_3 \# \ldots \# Y_{2j} \# Y_{2j+1} \# \} \]

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

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

But, \( L_2 = L( G_2 ) = \{ \#X_0\#X_1\#X_2\#X_3\#X_4 \#\ldots\# X_{2k-1}\#X_{2k}\#Z_0\# \} \)

where \( X_{2i-1} \Rightarrow X_{2i}, 1 \leq i \leq k. \)

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

\[ L = L_1 \cap L_2 \] describes correct traces (checked even/odd and odd/even). If \( Z_0 \) is chosen to be a terminal configuration, then these are terminating traces. If we pick a fixed \( X_0 \), then \( X_0 \) is a halting configuration iff \( L \) is non-empty. This is an independent proof of the undecidability of the non-empty intersection problem for CFGs and the non-emptiness problem for CSGs.
What’s Undecidable?

• We cannot decide if the set of valid terminating traces of an arbitrary machine M is non-empty.

• We cannot decide if the complement of the set of valid terminating traces of an arbitrary machine M is everything. In fact, this is not even semi-decidable.
$L = \Sigma^*?$

- If $L$ is regular, then $L = \Sigma^*?$ is decidable
  - Easy – Reduce to minimal deterministic FSA, $A_L$ accepting $L$. $L = \Sigma^*$ iff $A_L$ is a one-state machine, whose only state is accepting

- If $L$ is context free, then $L = \Sigma^*?$ is undecidable
  - Just produce the complement of a Turing Machine’s valid terminating traces
Finite Convergence for Concatenation of Context-Free Languages

Relation to Real-Time (Constant Time) Execution
Powers of CFLs

Let G be a context free grammar. Consider $L(G)^n$

Question1: Is $L(G) = L(G)^2$?

Question2: Is $L(G)^n = L(G)^{n+1}$, for some finite $n>0$?

These questions are both undecidable. Think about why question1 is as hard as whether or not $L(G)$ is $\Sigma^*$. Question2 requires much more thought.
L(G) = L(G)^2?

• The problem to determine if \( L = \Sigma^* \) is Turing reducible to the problem to decide if \( L \cdot L \subseteq L \), so long as \( L \) is selected from a class of languages \( C \) over the alphabet \( \Sigma \) for which we can decide if \( \Sigma \cup \{\lambda\} \subseteq L \).

• Corollary 1:
The problem “is \( L \cdot L = L \), 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 CSL 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)
Finite Power Problem

• The problem to determine, for an arbitrary context free language L, if there exist a finite n such that $L^n = L^{n+1}$ is undecidable.

• $L_1 = \{ C_1 \# C_2^R \$ \mid C_1, C_2 \text{ are configurations} \}$,

• $L_2 = \{ C_1 \# C_2^R \$ C_3 \# C_4^R \ldots \$ C_{2k-1} \# C_{2k}^R \$ \mid \text{where } k \geq 1 \text{ and, for some } i, 1 \leq i < 2k, C_i \Rightarrow_M C_{i+1} \text{ is false} \}$,

• $L = L_1 \cup L_2 \cup \{\lambda\}$. 
Undecidability of $\exists n \ L^n = L^{n+1}$

- $L$ is context free.
- Any product of $L_1$ and $L_2$, which contains $L_2$ at least once, is $L_2$. For instance, $L_1 \cdot L_2 = L_2 \cdot L_1 = L_2 \cdot L_2 = L_2$.
- This shows that $(L_1 \cup L_2)^n = L_1^n \cup L_2$.
- Thus, $L^n = \{\lambda\} \cup L_1 \cup L_1^2 \ldots \cup L_1^n \cup L_2$.
- Analyzing $L_1$ and $L_2$ we see that $L_1^n \cup L_2 \neq L_2$ just in case there is a word $C_1 \# C_2^R \$ C_3 \# C_4^R \ldots \$ C_{2n-1} \# C_{2n}^R \$ in $L_1^n$ that is not also in $L_2$.
- But then there is some valid trace of length $2n$.
- $L$ has the finite power property iff $M$ executes in constant time.
Turing Machine Traces

• A valid trace
  \[ C_1 \# C_2^R \$ C_3 \# C_4^R \ldots \$ C_{2k-1} \# C_{2k}^R \$, where \( k \geq 1 \) and \( C_i \Rightarrow_M C_{i+1} \), for \( 1 \leq i < 2k \). Here, \( \Rightarrow_M \) means derive in \( M \), and \( C^R \) means \( C \) with its characters reversed.

• An invalid trace
  \[ C_1 \# C_2^R \$ C_3 \# C_4^R \ldots \$ C_{2k-1} \# C_{2k}^R \$, where \( k \geq 1 \) and for some \( i \), it is false that \( C_i \Rightarrow_M C_{i+1} \).
What’s Context Free?

• Given a Turing Machine M
  – The set of invalid traces of M is Context Free
  – The set of valid traces is Context Sensitive
  – The set of valid terminating traces is Context Sensitive
  – The complement of the valid traces is Context Free
  – The complement of the valid terminating traces is Context Free
Partially correct traces

$L_1 = L(G_1) = \{ #Y_0 # Y_1 # Y_2 # Y_3 # ... # Y_{2j} # Y_{2j+1} # \}$
where $Y_{2i} \Rightarrow Y_{2i+1}$, $0 \leq i \leq j$.
This checks the even/odd steps of an even length computation.

But, $L_2 = L(G_2) = \{ #X_0 # X_1 # X_2 # X_3 # X_4 # ... # X_{2k-1} # X_{2k} # Z_0 # \}$
where $X_{2i-1} \Rightarrow X_{2i}$, $1 \leq i \leq k$.
This checks the odd/steps of an even length computation.

$L = L_1 \cap L_2$ describes correct traces (checked even/odd and odd/even). If $Z_0$ is chosen to be a terminal configuration, then these are terminating traces. If we pick a fixed $X_0$, then $X_0$ is a halting configuration iff $L$ is non-empty. This is an independent proof of the undecidability of the non-empty intersection problem for CFGs and the non-emptiness problem for CSGs.
What’s Undecidable?

- We cannot decide if the set of valid terminating traces of an arbitrary machine M is non-empty.
- We cannot decide if the complement of the set of valid terminating traces of an arbitrary machine M is everything. In fact, this is not even semi-decidable.
If \( L \) is regular, then \( L = \Sigma^*? \) is decidable
- Easy – Reduce to minimal deterministic FSA, \( a_L \) accepting \( L \). \( L = \Sigma^* \) iff \( a_L \) is a one-state machine, whose only state is accepting

If \( L \) is context free, then \( L = \Sigma^*? \) is undecidable
- Just produce the complement of a Turing Machine’s valid terminating traces
Propositional Calculus

Axiomatizable Fragments
Propositional Calculus

- Mathematical of unquantified logical expressions
- Essentially Boolean algebra
- Goal is to reason about propositions
- Often interested in determining
  - Is a well-formed formula (wff) a tautology?
  - Is a wff refutable (unsatisfiable)?
  - Is a wff satisfiable? (classic NP-complete)
Tautology and Satisfiability

- The classic approaches are:
  - Truth Table
  - Axiomatic System (axioms and inferences)
- Truth Table
  - Clearly exponential in number of variables
- Axiomatic Systems Rules of Inference
  - Substitution and Modus Ponens
  - Resolution / Unification
Proving Consequences

- Start with a set of axioms (all tautologies)
- Using substitution and MP
  \[(P, P \supset Q \supset Q)\]
  derive consequences of axioms (also tautologies, but just a fragment of all)
- Can create complete sets of axioms
- Need 3 variables for associativity, e.g.,
  \[(p1 \lor p2) \lor p3 \supset p1 \lor (p2 \lor p3)\]
Some Undecidables

• Given a set of axioms,
  – Is this set complete?
  – Given a tautology $T$, is $T$ a consequent?

• The above are even undecidable with one axiom and with only 2 variables. I will show this result shortly.
Refutation

• If we wish to prove that some wff, F, is a tautology, we could negate it and try to prove that the new formula is refutable (cannot be satisfied; contains a logical contradiction).

• This is often done using resolution.
Resolution

• Put formula in Conjunctive Normal Form (CNF)
• If have terms of conjunction 
  (P ∨ Q), (R ∨ ~Q)
  then can determine that (P ∨ R)
• If we ever get a null conclusion, we have refuted the proposition
• Resolution is not complete for derivation, but it is for refutation
Axioms

• Must be tautologies
• Can be incomplete
• Might have limitations on them and on WFFs, e.g.,
  – Just implication
  – Only n variables
  – Single axiom
Simulating Machines

• Linear representations require associativity, unless all operations can be performed on prefix only (or suffix only)
• Prefix and suffix based operations are single stacks and limit us to CFLs
• Can simulate Post normal Forms with just 3 variables.
Diadic PIPC

- Diadic limits us to two variables
- PIPC means Partial Implicational Propositional Calculus, and limits us to implication as only connective
- Partial just means we get a fragment
- Problems
  - Is fragment complete?
  - Can F be derived by substitution and MP?
Living without Associativity

• Consider a two-stack model of a TM
• Could somehow use one variable for left stack and other for right
• Must find a way to encode a sequence as a composition of forms – that’s the key to this simulation
Composition Encoding

• Consider \((p \bowtie p)\), \((p \bowtie (p \bowtie p))\),
  \((p \bowtie (p \bowtie (p \bowtie p)))\), …
  – No form is a substitution instance of any of the other, so they can’t be confused
  – All are tautologies

• Consider \(((X \bowtie Y) \bowtie Y)\)
  – This is just \(X \lor Y\)
Encoding

- Use \((p \supset p)\) as form of bottom of stack
- Use \((p \supset (p \supset p))\) as form for letter 0
- Use \((p \supset (p \supset (p \supset p)))\) as form for 1
- Etc.

- String 01 (reading top to bottom of stack) is
  
  \[- ( ( (p \supset p) \supset ( (p \supset p) \supset ( (p \supset p) \supset (p \supset p) ) ) ) ) ) \supset
  ( ( (p \supset p) \supset ( (p \supset p) \supset ( (p \supset p) \supset (p \supset p) ) ) ) ) \supset
  ( (p \supset p) \supset ( (p \supset p) \supset ( (p \supset p) \supset (p \supset p) ) ) ) ) )\]
Encoding

$I(p)$ abbreviates $[p ⊨ p]$

$\Phi_0(p)$ is $[p ⊨ I(p)]$ which is $[p ⊨ [p ⊨ p]]$

$\Phi_1(p)$ is $[p ⊨ \Phi_0(p)]$

$\xi_1(p)$ is $[p ⊨ \Phi_1(p)]$

$\xi_2(p)$ is $[p ⊨ \xi_1 (p)]$

$\xi_3(p)$ is $[p ⊨ \xi_2 (p)]$

$\psi_1(p)$ is $[p ⊨ \xi_3 (p)]$

$\psi_2(p)$ is $[p ⊨ \psi_1 (p)]$

…

$\psi_m(p)$ is $[p ⊨ \psi_{m-1} (p)]$
Creating Terminal IDs

1. $[\xi_1 I(p_1) \lor I(p_1)]$.
2. $[\xi_1 I(p_1) \lor I(p_1)] \supset [\xi_1 I(p_1) \lor \Phi_1 I(p_1)]$.
3. $[\xi_1 I(p_1) \lor \Phi_i(p_2)] \supset [\xi_1 I(p_1) \lor \Phi_j \Phi_i(p_2)], \forall i, j \in \{0, 1\}$.
4. $[\xi_1 I(p_1) \lor p_2] \supset [\xi_2 \Phi_1 I(p_1) \lor p_2]$.
5. $[\xi_1 I(p_1) \lor p_2] \supset [\xi_3 \Phi_i I(p_1) \lor p_2], \forall i \in \{0, 1\}$.
6. $[\xi_2 \Phi_i(p_1) \lor p_2] \supset [\xi_2 \Phi_j \Phi_i(p_1) \lor p_2], \forall i, j \in \{0, 1\}$.
7. $[\xi_2 \Phi_i(p_1) \lor p_2] \supset [\xi_3 \Phi_j \Phi_i(p_1) \lor p_2], \forall i, j \in \{0, 1\}$.
8. $[\xi_3 \Phi_i(p_1) \lor p_2] \supset [\Psi_k \Phi_i(p_1) \lor p_2], \text{ whenever } q_k a_i \text{ is a terminal discriminant of } M.$
Reversing Print and Left

9. $[Ψ'_k Φ_i(p_1) \lor p_2] \Rightarrow [Ψ'_h Φ_j(p_1) \lor p_2]$, whenever $q_h a_j a_i q_k \in T$.

10a. $[Ψ'_k Φ_0 I(p_1) \lor I(p_1)] \Rightarrow [Ψ'_h Φ_0 I(p_1) \lor I(p_1)]$,
    b. $[Ψ'_k Φ_1 I(p_1) \lor I(p_1)] \Rightarrow [Ψ'_h Φ_0 I(p_1) \lor Φ_1(p_1)]$,
    c. $[Ψ'_k Φ_i I(p_1) \lor Φ_j(p_2)] \Rightarrow [Ψ'_h Φ_0 I(p_1) \lor Φ_i Φ_j(p_2)]$,
    d. $[Ψ'_k Φ_0 Φ_0 Φ_i(p_1) \lor I(p_2)] \Rightarrow [Ψ'_h Φ_0 Φ_i(p_1) \lor I(p_2)]$,
    e. $[Ψ'_k Φ_1 Φ_0 Φ_i(p_1) \lor I(p_2)] \Rightarrow [Ψ'_h Φ_0 Φ_i(p_1) \lor Φ_1 I(p_2)]$,
    f. $[Ψ'_k Φ_i Φ_0 Φ_j(p_1) \lor Φ_m(p_2)] \Rightarrow [Ψ'_h Φ_0 Φ_j(p_1) \lor Φ_i Φ_m(p_2)]$,
    $\forall i, j, m \in \{0, 1\}$ whenever $q_h 0 L q_k \in T$.

11a. $[Ψ'_k Φ_0 Φ_1(p_1) \lor I(p_2)] \Rightarrow [Ψ'_h Φ_1(p_1) \lor I(p_2)]$,
    b. $[Ψ'_k Φ_1 Φ_1(p_1) \lor I(p_2)] \Rightarrow [Ψ'_h Φ_1(p_1) \lor Φ_1 I(p_2)]$,
    c. $[Ψ'_k Φ_i Φ_1(p_1) \lor Φ_j(p_2)] \Rightarrow [Ψ'_h Φ_1(p_1) \lor Φ_i Φ_j(p_2)]$,
    $\forall i, j \in \{0, 1\}$ whenever $q_k 1 L q_k \in T$. 
Reversing Right

12a. $[\Psi_k \Phi_0 I(p_1) \lor I(p_1)] \supset [\Psi_h \Phi_0 I(p_1) \lor I(p_1)]$,

b. $[\Psi_k \Phi_0 I(p_1) \lor \Phi_0 \Phi_i(p_2)] \supset [\Psi_h \Phi_0 I(p_1) \lor \Phi_i(p_2)]$,

c. $[\Psi_k \Phi_1(p_1) \lor I(p_2)] \supset [\Psi_h \Phi_0 \Phi_1(p_1) \lor I(p_2)]$,

d. $[\Psi_k \Phi_0 \Phi_i(p_1) \lor I(p_2)] \supset [\Psi_h \Phi_0 \Phi_0 \Phi_i(p_1) \lor I(p_2)]$,

e. $[\Psi_k \Phi_0 \Phi_i(p_1) \lor \Phi_0 \Phi_j(p_2)] \supset [\Psi_h \Phi_0 \Phi_0 \Phi_i(p_1) \lor \Phi_j(p_2)]$,

f. $[\Psi_k \Phi_1(p_1) \lor \Phi_0 \Phi_i(p_2)] \supset [\Psi_h \Phi_0 \Phi_1(p_1) \lor \Phi_i(p_2)]$,

$\forall i, j \in \{0, 1\}$ whenever $q_h 0 R q_k \in T$.

13a. $[\Psi_k \Phi_0 I(p_1) \lor \Phi_1(p_2)] \supset [\Psi_h \Phi_1 I(p_1) \lor p_2]$, 

b. $[\Psi_k \Phi_1(p_1) \lor \Phi_1(p_2)] \supset [\Psi_h \Phi_1 \Phi_1(p_1) \lor p_2]$, 

c. $[\Psi_k \Phi_0 \Phi_1(p_1) \lor \Phi_1(p_2)] \supset [\Psi_h \Phi_1 \Phi_0 \Phi_i(p_1) \lor p_2]$

$\forall i \in \{0, 1\}$ whenever $q_h 1 R q_k \in T$. 