University of
Central
Florida

Discrete II
Theory of Computation

Charles E. Hughes

COT 4210 – Fall 2017 Notes

Who, What, Where and When

- **Instructor**: Charles Hughes;
  Harris Engineering 247C; 823-2762
  (phone is not a good way to get me);
  charles.e.hughes@knights.ucf.edu
  (e-mail is a good way to get me)
  Please use Subject: COT4210
- **Web Page**: http://www.cs.ucf.edu/courses/cot4210/Spring2017
- **Meetings**: TR 1:30PM – 2:45PM, CB2-105;
  30 class periods, each 75 minutes long.
  Office Hours: TR 3:15PM – 4:30PM in HEC-247C
- **GTA**: Anthony Wehrer
  awehrer@knights.ucf.edu
  Please use Subject: COT4210
  Office Hours: W 3:00PM-4:15PM; F 4:00PM-5:15PM; HEC-308

Text Material

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

Expectations

- **Prerequisites**: COT3100 (discrete structure I); COP3503 (undergraduate algorithm design and analysis).
- **Assignments**: Assignments will be graded but there may also be ungraded practice problems.
- **Quizzes:** There may be occasional quizzes. These will include the same types of questions asked on assignments.
- **Exams**: Two (2) midterms and a final.
- **Material**: I will draw heavily from the text by Sipser (Chapters 1-5 and 7). Some material will also come from Hopcroft. Class notes and in-class discussions are, however, comprehensive and cover models, closure properties and undecidable problems that may not be addressed in either of these texts. Note, however, that the Notes are often guidelines to topics in the text, so do not ignore Sipser.

Goals of Course

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

Expected Outcomes

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

Keeping Up

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

Rules to Abide By

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

Exam Grading

- Overall exam grade involves combining the two midterms, weighing the better of these two higher than the weaker, and then combining that aggregate score with the final where the weight of either the better of the aggregate midterm or the final gets increased by 50.
- If you do the numbers you will see that It is necessary to have at least an A- average on the combined exams to get an A for the course.

Important Dates

- Exam#1 – Tentatively Thursday, September 28
- Withdraw Deadline – Monday, October 30
- Exam#2 – Tentatively Thursday, November 2
- Final – Tuesday, Dec. 5, 1:00PM–3:50PM
- Days off: 8/31 (Football); 11/23 (Thanksgiving)
- Exam #1/#2 dates are subject to change with appropriate notice. Final exam is, of course, fixed in stone.

Evaluation (tentative)

- Mid Terms – 100 points each (combined for 200)
- Final Exam – 175 points
- Quizzes and Assignments – 75 points
- Bonus – best exam (combined midterm or final) weighed +50 points
- Total Available: 500
- Grading will be A ≥ 90%, A- ≥ 88%,
  B+ ≥ 85%, B ≥ 80%, B- ≥ 78%,
  C+ ≥ 75%, C ≥ 70%, C- ≥ 60%,
  D ≥ 50%, F < 50%

Navigating Notes

- When a slide is presenting a problem set, I will highlight the slide title in **Red**
- When a topic is not in the text, I will highlight the slide title in **Green**
- When a topic is covered either in part or only in exercises in the text, I will highlight the slide title in **Blue**

Assignment # 1 Includes Financial Aid Related Activity

1. Send an e-mail to me.
   The subject must be **COT4210**.
   Send it to charles.e.hughes@knights.ucf.edu
   I will use that for all class communication.
   Cc: GTA awehrer@knights.ucf.edu
   In the message, tell me where and when you took Discrete Structures I or its equivalent. Also, tell me what days/times you are **NOT** free to make office hours.
2. Prove the following: Let R be an equivalence relation over some universe U, and let C_a be the class of all elements in U equivalent to the element a, i.e., $C_a = \{x \mid x \in U \;\&\&\; a\,R\,x\}$, and C_b be the class of all elements in U equivalent to the element b , i.e.,
   $C_b = \{x \mid x \in U \;\&\&\; b\,R\,x\}$.
   Prove that either $C_a = C_b$ or $C_a \cap C_b = \Phi$.
   The assignment needs to be submitted through Webcourses.

Complete and submit both parts by Midnight Friday, 8/25.

Sets, Sequences, Relations, Functions, Cardinality, Graphs and Languages

Mostly from Chapter 0 of Sipser

Sets

- *Sets* are unordered collections of distinct objects.
- Sets can be defined or specified in many ways:
  - By explicitly enumerating their members or elements
    e.g. S = { a, b, c}
    Note: If S' = { b, c, a}, then S and S' denote the same set (that is, S' = S)
  - By specifying a condition for membership
    S = { x ∈ Δ | P(x) }, reads "S is the set of all x in Δ such that P(x) is true"
    P is called a "predicate" (a function from set Δ to {true, false})
    E.g. S = { x ∈ UCF_Students | x is a CS_major }
- The empty set is denoted, Ø, and is the set with no members; that is,
  Ø = { }. Also, the predicate, x ∈ Ø is always false!
- *Multisets* or *Bags* are unordered collections of objects where we keep track of repeated elements (usually with a count per element)

More on Sets

- If S ≠ Ø, then there exists an x for which x ∈ S is true; this predicate is read "x is an element of S" or "x is a member of S". The symbol "∈" denotes the member relation. x ∉ S is true when x is not in S.
- We use normal set operation of union (A ∪ B), intersection (A ∩ B) and complement ~A (usually A with a bar on it).
- If A and B are sets, then we write "A ⊆ B" to mean that A is a subset of B. This means that for all x ∈ A, x ∈ B. Or, ∀x [x ∈ A ⇒ x ∈ B].
- The expression, "A ⊊ B" means that A is a proper subset of B. Mathematically, ∀x [x ∈ A ⇒ x ∈ B] and ∃y [y ∈ B and y ∉A]
- The cross (Cartesian) product of two sets A and B is denoted, A × B, and is the set defined as follows: A × B = { (a,b) | a ∈ A and b ∈ B } . "(a,b)" is an expression composed from elements, a,b, selected arbitrarily from sets A and B, respectively. If A ≠ B, then A × B ≠ B × A.
  Note: (a,b) is a sequence not a set. See next slide.

Sequences

- While sets have no order and no repeated elements, *sequences* have order and can contain repeats at differing positions in the order.
  - The set {5,2,5} = {5,2} = {2,5}
  - The sequence (5,2,5) ≠ (2,5,5) ≠ (5,5,2) ≠ (5,2) ≠ (2,5)
- Actually, there is a notion of a *multiset* or *bag* that we sometimes use. It has no order, but repeated elements are allowed. Since position is irrelevant, we just record each unique elements with a count.
- We can talk about the *k-th element* of a sequence, but not of a set or multiset.
- Finite sequences are often called *tuples*. Those of length k are *k-tuples*. A 2-tuple is also called a *pair*.

Relations

- A *relation*, r, is a mapping from some set A to some set B;

  We write, r: A → B, and we mean that r assigns to every member of A a subset of B; that is, for every a $\in$ A, r(a) $\subseteq$ B and r(a) $\neq$ Ø.

  A relation, r, can also be defined in terms of the cross product of A and B:
  r $\subseteq$ A $\times$ B such that for every a $\in$ A there is at least one b $\in$ B such that (a, b) $\in$ r.

- We say that a relation, r, from A to B is a *partial relation* if and only if for some a $\in$ A, r(a) = Ø = { }.

More on Relations

- A *predicate* or *property* is a function with range {TRUE, FALSE}
- A property with a domain of *n*-tuples A^n is an *n*-ary relation
- Binary relations are common, and like binary functions, we use infix notations for them
- Let *R* be a binary relation on A^2. *R* is:
  - *Reflexive* if $\forall\ x \in a,\ x\ R\ x$
  - *Symmetric* if $x\ R\ y \rightarrow y\ R\ x$
  - *Transitive* if $(\ x\ R\ y,\ y\ R\ z\ ) \rightarrow x\ R\ z$
  - An *equivalence* relation if it is reflexive, symmetric and transitive

Functions

- Functions are special types of relations. Specifically, a relation f: A→ B, is said to be a (total) function from A to B if and only if, for every a ∈ A, f(a) has exactly one element; that is, |f(a)| = 1.
- If f is a *partial* function from A to B, then f may not be defined for every a ∈ A. In this case we write |f(a)| ≤ 1, for every a in A; note that |f(a)| = 0 if and only if f(a) = Ø, and we say the function is *undefined* at a.
  Note: Text calls the set of possible inputs a function's *domain*. We will often use domain for the set of input values on which f is defined, referring to the input set as the universe of discourse. If a function is *total* (defined everywhere) then there is no terminology difference.
- A function, f, is said to be one-to-one (1-1) if and only if x ≠ y implies f(x) ≠ f(y). A total function that is one-to-one is sometimes called an *injection.*
- A function, f: A→ B, is said to be onto if and only if for every y ∈ B there is an x ∈ A such that y = f(x).
  Note: technically we should write {y} = f(x), since functions are relations, however, the more convenient and less baroque notation is used when dealing with functions. Total functions that are onto are called *surjections*. Ones that are 1-1 and onto are called *bijections*.

Ordinal and Cardinal Numbers

Definition. *Ordinal numbers* are symbols used to designate relative position in an ordered collection. The ordinals correspond to the natural numbers: 0, 1, 2, … The set of all natural (ordinal) numbers is denoted, ***N***. (Note: Here we include 0 as a natural number.)

A fundamental concept in set theory is the size of a set, S. We begin with a definition.

Definition. Let S be any set. We associate with S, the unique symbol |S| called its *cardinality*. Symbols of this kind are called *cardinal numbers* and denote the size of the set with which they are associated.

$|Ø| = 0$ (the cardinal number defining the size of the empty set is the ordinal, 0)

If $S = \{0, 1, 2, 3, …, n-1\}$, for some natural number $n>0$, then $|S|=n$.

To summarize, the cardinality of any finite set (including the empty set) is simply the ordinal number that specifies the number of elements in that set.

More on Cardinality

To determine the relative size of two sets, we need the following definitions:

Definition. If A and B are two sets, then $|A| \leq |B|$ if and only if there exists an injection, f, from A to B; f is a 1-1 function from A into B.

Definition. If A and B are two sets, then $|A| = |B|$ if and only if $|A| \leq |B|$ and $|B| \leq |A|$. We may also say that $|A| = |B|$ if and only if there is a bijection, f, from A to B; f is a 1-1 function from A onto B.

Definition. If A and B are two sets, then $|A| < |B|$ if and only if $|A| \leq |B|$ and $|A| \neq |B|$.

Definition. A set S is said to be finite if and only if $|S| \in \boldsymbol{N}$; otherwise, S is said to be infinite. A set S is said to be countable if and only if S is finite or $|S| = |\boldsymbol{N}|$; otherwise S is said to be uncountable. We discuss cardinality in more details later.

Infinities

By the definitions above, there are many infinite sets with which you are familiar.

For example:

$\boldsymbol{N}$ (the set of Natural numbers), $\boldsymbol{Z}$ (the set of Integers), $\boldsymbol{Z}^+$ (the set of Positive Integers), $\boldsymbol{Q}$ (the set of Rational numbers) and $\boldsymbol{R}$ (the set of Real numbers).

But, are all these infinite sets the same size??

Brash statement: $|\boldsymbol{N}| = |\boldsymbol{Z}^+| = |\boldsymbol{Z}| = |\boldsymbol{Q}| < |\boldsymbol{R}|$.

Power Set

Definition. Let S be a set, then the power set of S, denoted $\mathcal{P}(S)$ or 2^S, is defined by
$\mathcal{P}(S) = \{ A \mid A \subseteq S \}$.

Examples.

$\mathcal{P}(\emptyset) = \{\emptyset\}$,

$\mathcal{P}(\{1,2,3\}) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$

$\mathcal{P}(\mathbf{N}) = \{\emptyset, \{0\}, \{1\}, \{2\}, \{3\}, \ldots$
$, \{0,1\}, \{0,2\}, \{0,3\}, \ldots$
$, \{0,1,2\}, \ldots$
$\ldots \{ \mathbf{N} \} \}$

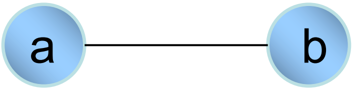
Undirected Graphs

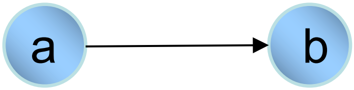
- An **undirected Graph G** is defined by a pair **(V, E)**
- **V**: Finite Set of **Nodes/Vertices**
- **E**: { <a,b> | a,b∈**V** are called **Edges/Arcs**}
  - **E**⊆**V**×**V** such that <a,b>∈**E** implies <b,a>∈**E**
- **Degree** of node is number of edges at that node (number of nodes it relates to)
- Graphs can be **labeled**, as we did above on the nodes, or unlabeled.
- Labels can go on nodes, edges or both.

More on Graphs

- A **subgraph** H of a graph G is a subset of the nodes of G with all edges retained from G that involve node pairs in H.
- A **path** is a sequence of nodes connected by edges.
- A graph is **connected** if every two nodes are connected by a path.
- A **cycle** is a path that starts and ends in the same node.
- A **simple cycle** is a path that involves at least three nodes and starts and ends in the same node. (excludes self loop)
- A **tree** is a graph that is connected and has no simple cycles.
- A tree may contain a special node called the **root**.
- The nodes of degree 1 in a tree, excepting the root, are called **leaves**.
- The set of leaves of a tree are called the **frontier**.
- If the edges have direction then a graph is called **directed**

Directed vs Undirected

- If directed, we differentiate **in-degree** (edges into node) from **out-degree (edges out of node)**.
- Undirected 

Directed 

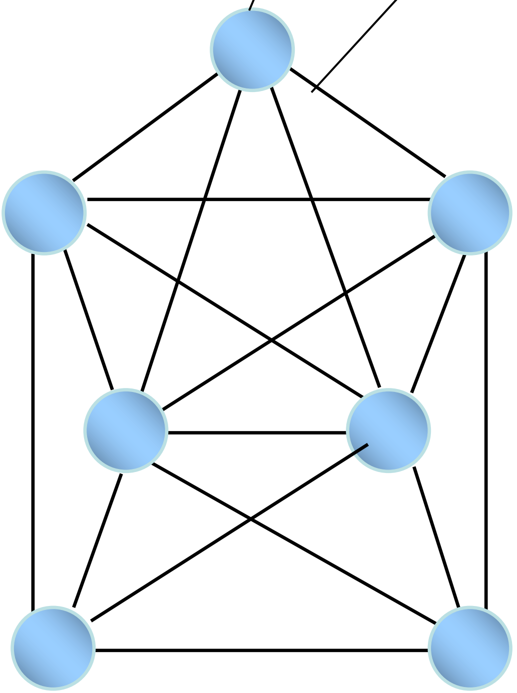
Graph G = (V, E)

V: Finite Set of Nodes/Vertices
E: V × V → V are Edges/Arcs

Undirected

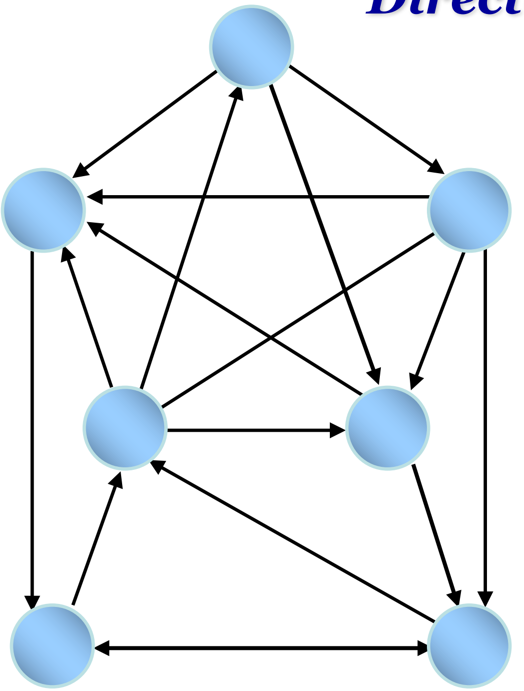
Nodes / Vertices

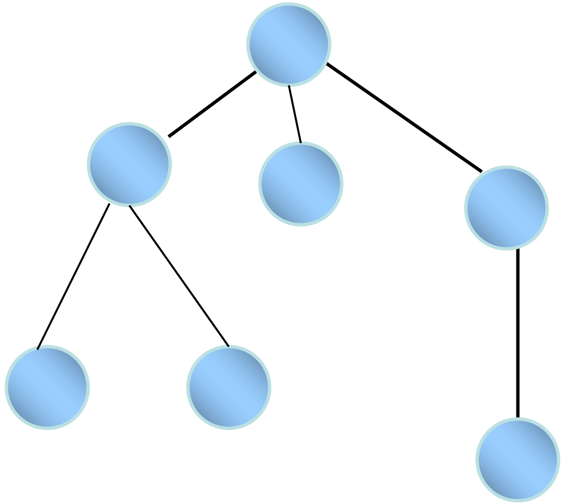
Edges / Arcs



$(v_i, v_j) = (v_j, v_i)$

Directed





Tree has no simple cycles and often has a root

Alphabets and Strings

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

Languages

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

Operations on Strings

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

Properties of Languages

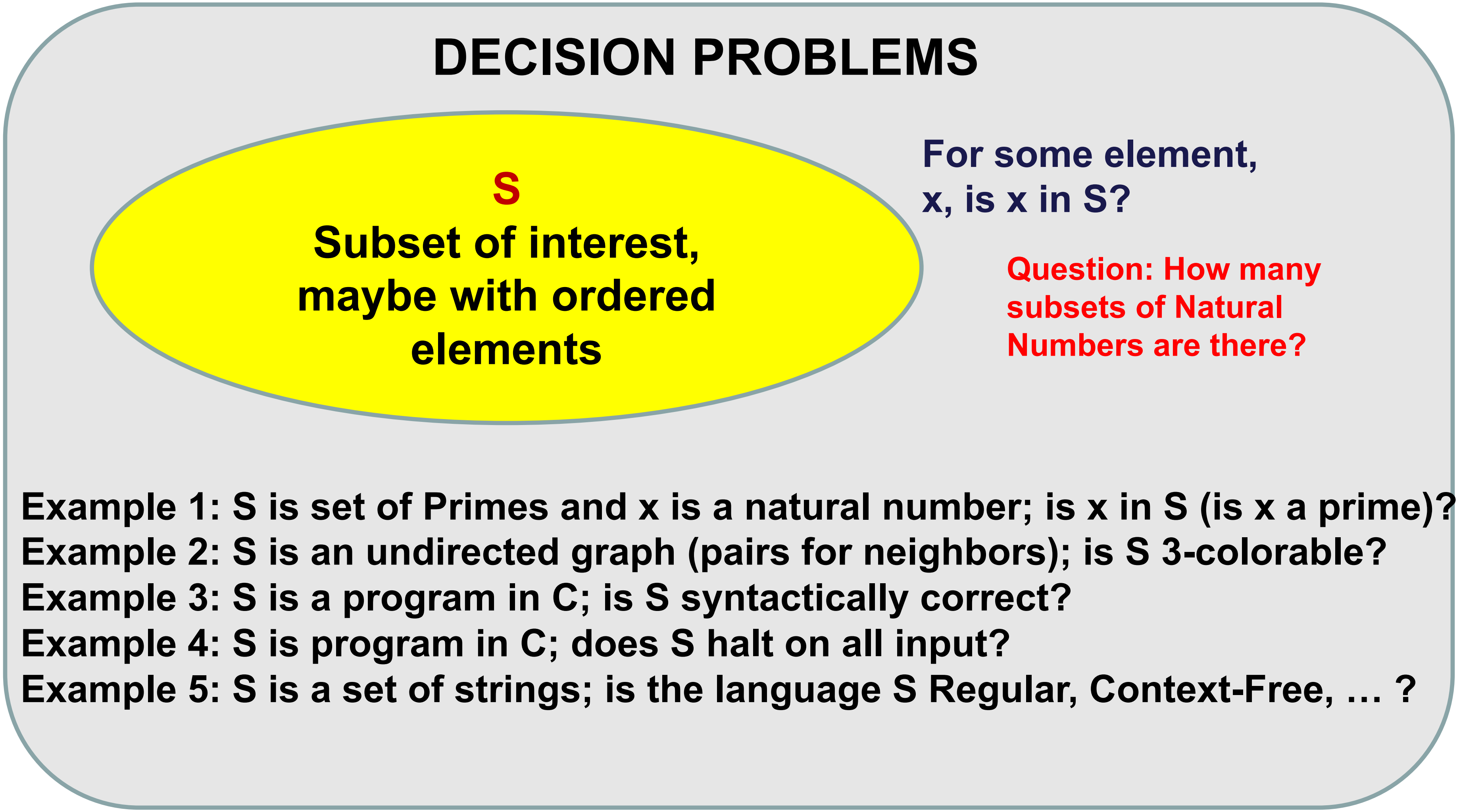
- Let L, M and N be languages over Σ, then:
  - $\emptyset \cdot L = L \cdot \emptyset = \emptyset$
  - $\{\lambda\} \cdot L = L \cdot \{\lambda\} = L$
  - $L \cdot (M \cup N) = L \cdot M \cup L \cdot N$ and $(M \cup N) \cdot L = M \cdot L \cup N \cdot L$
    - Concatenation does **NOT** distribute over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

Recognizer and Generators

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

UNIVERSE OF DISCOURSE

USUALLY STRINGS OR NATURAL NUMBERS



DECISION PROBLEMS

S

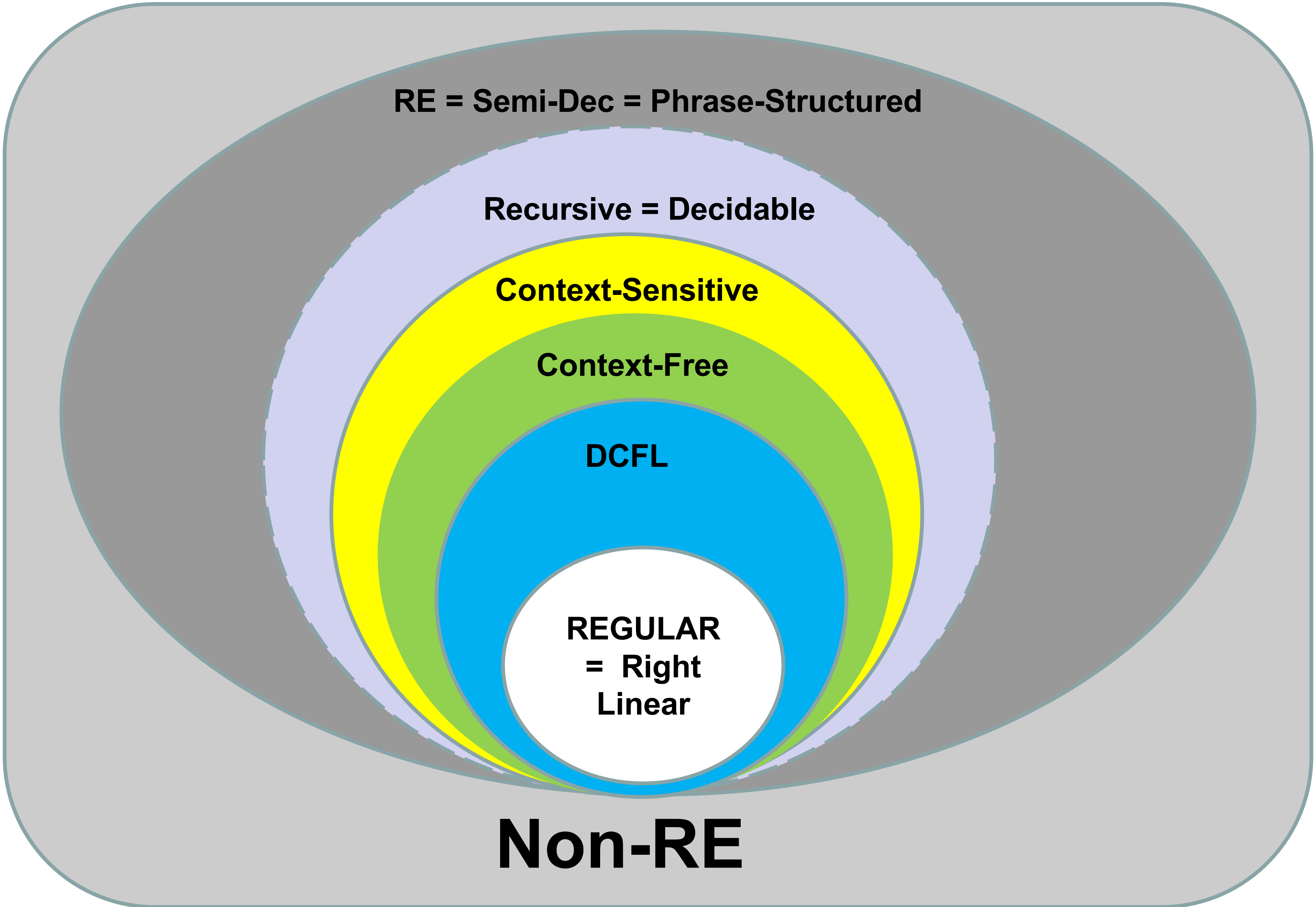
Subset of interest, maybe with ordered elements

For some element, x, is x in S?

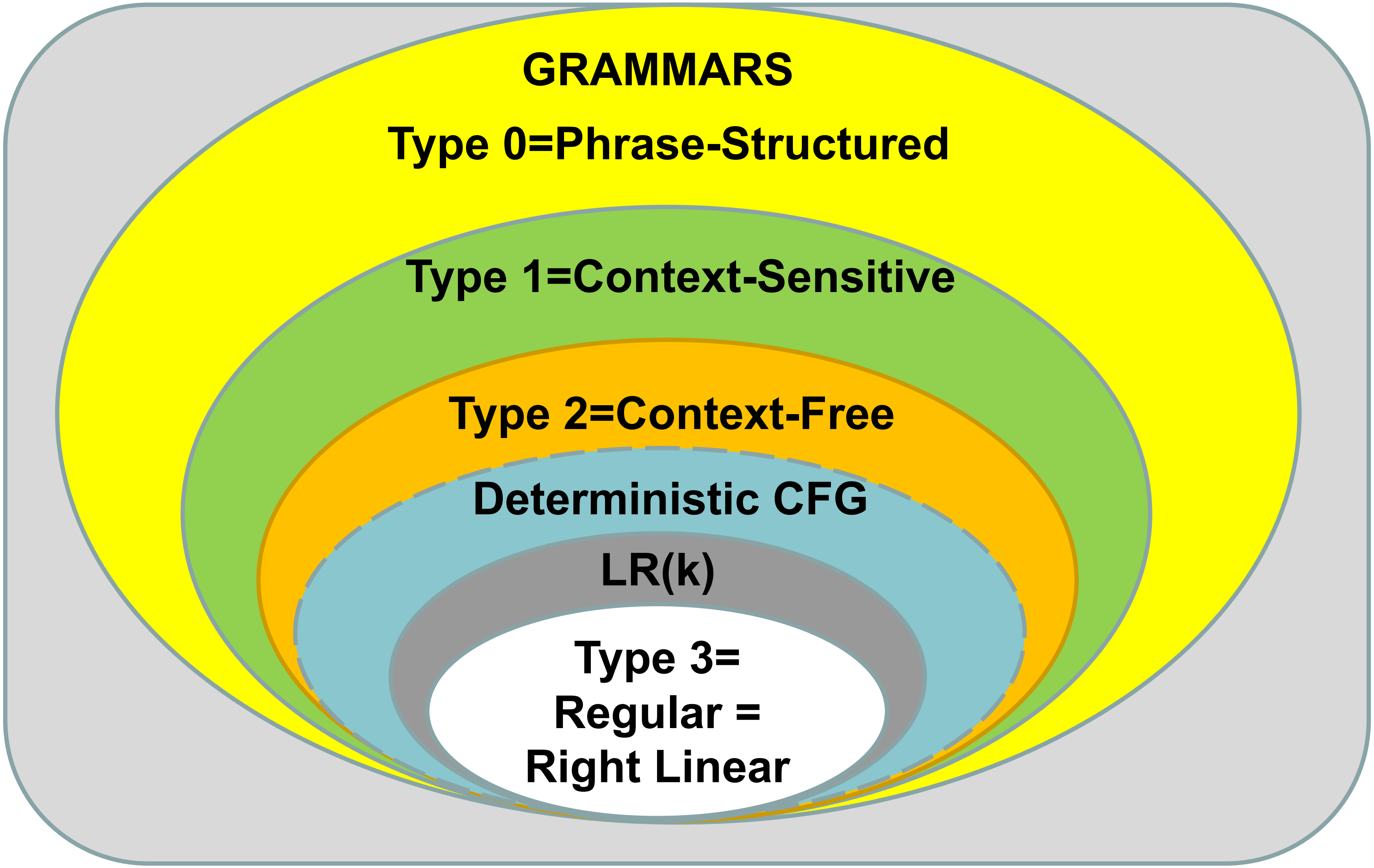
Question: How many subsets of Natural Numbers are there?

Example 1: S is set of Primes and x is a natural number; is x in S (is x a prime)?
Example 2: S is an undirected graph (pairs for neighbors); is S 3-colorable?
Example 3: S is a program in C; is S syntactically correct?
Example 4: S is program in C; does S halt on all input?
Example 5: S is a set of strings; is the language S Regular, Context-Free, … ?

UNIVERSE OF LANGUAGES



REWRITING SYSTEMS



GRAMMARS

Type 0=Phrase-Structured

Type 1=Context-Sensitive

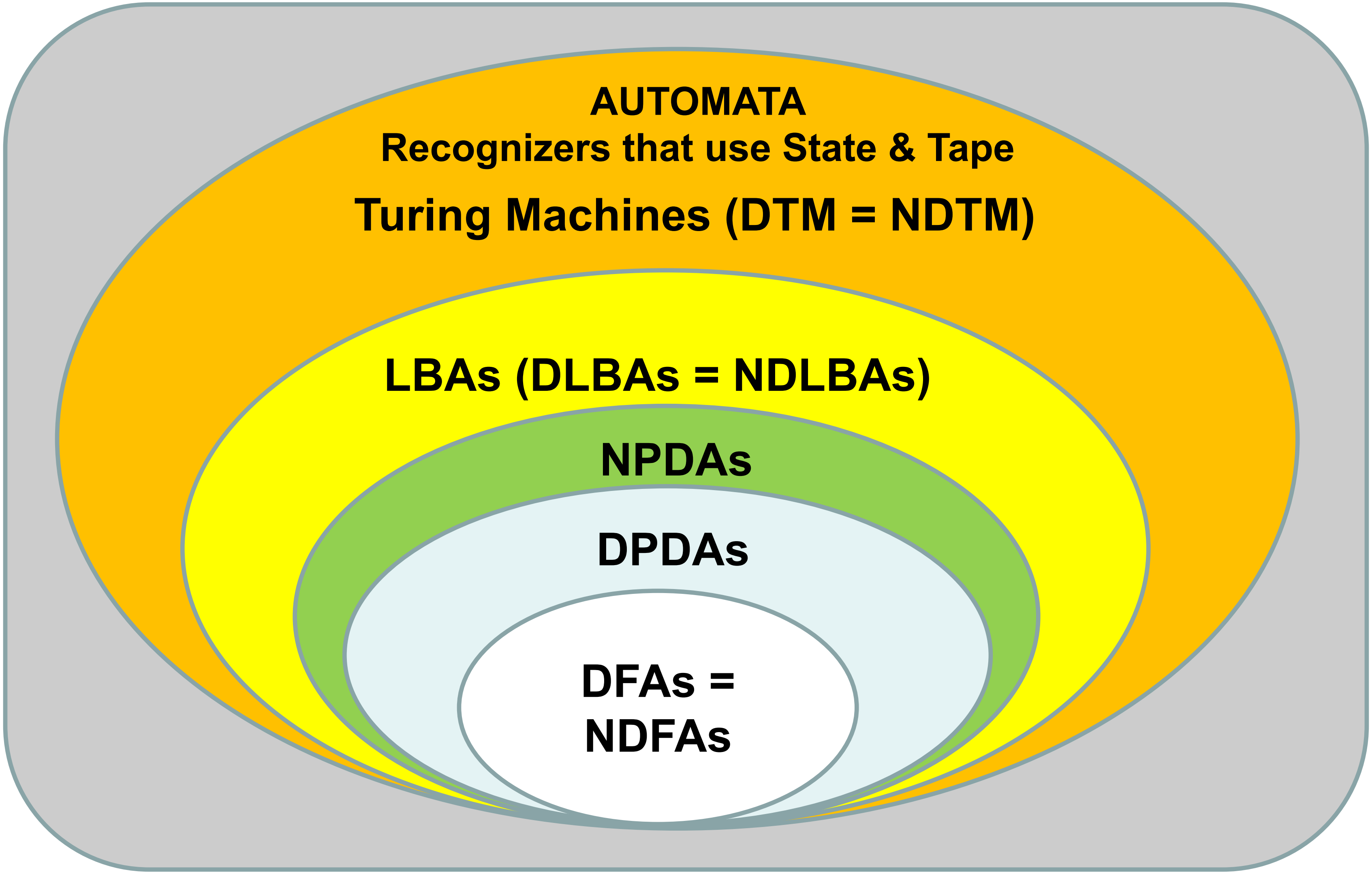
Type 2=Context-Free

Deterministic CFG

LR(k)

Type 3=
Regular =
Right Linear

MODELS OF COMPUTATION



AUTOMATA

Recognizers that use State & Tape

Turing Machines (DTM = NDTM)

LBAs (DLBAs = NDLBAs)

NPDAs

DPDAs

DFAs =
NDFAs

Of these models, only TMs can do general computation

Proofs

Terminology

- **Definitions** describe the mathematical objects and ideas we want to work with
- **Statements** or **assertions** are things we say about mathematics; they can be true or false
- **Proofs** are unassailable logical demonstrations that statements are true
- **Theorems** are statements that have been proven true
- **Lemmas** are theorems that are not interesting on their own but are useful for proving other theorems
- **Corollaries** are follow-on theorems that are easy to prove once you prove their parent theorems

Types of Proofs

- **Direct Argument**
  - Use assertions from theorem statement, known true properties and valid rules of inference
- **Construction**
  - Prove something exists by showing how to make it
- **Contradiction**
  - Prove something is true by showing it can't be false
  - One specific kind of proof by contradiction uses a technique called diagonalization
- **Weak Induction**
  - Show that a statement is true for some base case (often 0 or 1)
  - Show that *if* it's true for the case of some $i \geq$ base case, it's also true for the case of $i + 1$
- **Strong Induction**
  - Show that it's true for for some base case (often 0 or 1)
  - Show that *if* it's true for all cases where $\leq i$, where $i \geq$ base case, it's true for the case of $i + 1$

Sample Proof by Induction

Prove, if n is a positive whole number and $n \geq 4$, then $2^n \geq n^2$. Hint: use induction with a base of $n=4$.

Proof by Induction:

Base Case: $n = 4$: $2^4 \geq 4^2$ since $16 \geq 16$.

Induction Hypothesis: Assume $2^k \geq k^2$, for some $k \geq 4$.

Induction Step: Prove $2^{(k+1)} \geq (k+1)^2$

First, we observe that $k^2 \geq 2k+1$ when $k \geq 3$.

Consider $k=m+1$, where $k \geq 3$; and so $m \geq 2$

$k^2 = (m+1)^2 = m^2 + 2m+1 \geq 4 + 2m+1 > 2m+3 = 2(m+1) + 1 = 2k+1$.

Using this,

$2^{(k+1)} = 2^k * 2 = 2^k+2^k \geq k^2 + k^2 \geq k^2 + 2k + 1 = (k+1)^2$

QED

Sample Proof by Contradiction

Prove, if p and q are distinct prime numbers, then $\sqrt{(p/q)}$ is irrational. Assume $\sqrt{(p/q)}$ is rational where p and q are distinct primes. Let a/b be the reduced fraction (no common prime factors) that equals $\sqrt{(p/q)}$.

$\sqrt{(p/q)} = a/b$	: assumption (note $a \neq b$, as $p \neq q$)
$p/q = a^2/b^2$	: square both sides
$p = a^2$ and $q = b^2$	: since p and q have no common prime factors, and a and b have no common prime factors.

But this is not possible because p and q are prime numbers and so cannot have multiple factors (e.g., $a \times a$, in the case of p). This contradicts our original assumption that $\sqrt{(p/q)}$ is rational, so it must be irrational. QED

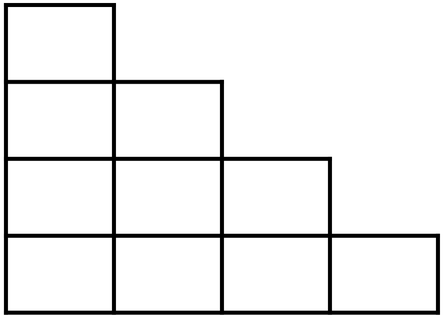
Practice Problems

Practice

1. Prove or disprove that, for sets A and B,
   A=B if and only if $(A \cap \sim B) \cup (A \cap B) = A$.
2. Prove that, for Boolean (T/F) variables P and Q,
   $((P \Rightarrow Q) \Rightarrow Q) \Leftrightarrow (P \vee Q)$
   $\vee$ is logical or; $\Rightarrow$ is logical implication; $\Leftrightarrow$ is logical equivalence
3. Prove: If S is any finite set with $|S| = n$, then
   $|S \times S \times S| \leq |\mathcal{P}(S)|$, for all $n \geq N$, where N is some constant, the minimum value of which you must discover and use as the basis for your proof.
4. Let L be a language over {a,b} where every string is of even length and is of the form WX, where $|W|=|X|$ but $W \neq X$. Design and present an algorithm that recognized strings in L using no unbounded amount of storage (no stacks, no queues). This means that any memory required must be of a fixed size independent of the length of an input string. Note: You cannot play the game of using unbounded recursion, as each call consumes stack space.
5. Show that, for any language L, if L has finite cardinality and contains some string of length > 0, then there cannot exist an N>1 such that $L^N = L^{N+1}$.

Assignment # 2

1. Consider the following function, **p**, from ℵ to ℵ.
   $p(0) = 0; p(x+1) = p(x) + x + 2$
   Prove inductively that $p(x) = (x^2 + 3x)/2$
2. Consider the following function, **q**, from ℵ to ℵ.
   $q(0) = 0; q(y+1) = q(y) + y + 1$
   Prove inductively that $q(y) = (y^2 + y)/2$
3. Consider the two variable function, **t**, from ℵ × ℵ to ℵ
   $t(x,y) = (x^2 + 3x + 2xy +y + y^2)/2$
   This embodies **p** and **q**, in that $p(x) = t(x,0)$ and $q(y) = t(0,y)$.
   Fill in the following matrix with values of $t(x,y)$ along the first four left to right diagonals (x is horizontal axis; y is vertical axis). Explain in a few sentences what the pattern is and how you could continue to fill in diagonals without ever looking back at the formula.



Due Thursday, August 31 at 1:30PM (use Webcourses to turn in)

Computability and Complexity

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

The study of really hard but computable problems

Goals of Computability

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

Hilbert, Russell and Whitehead

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

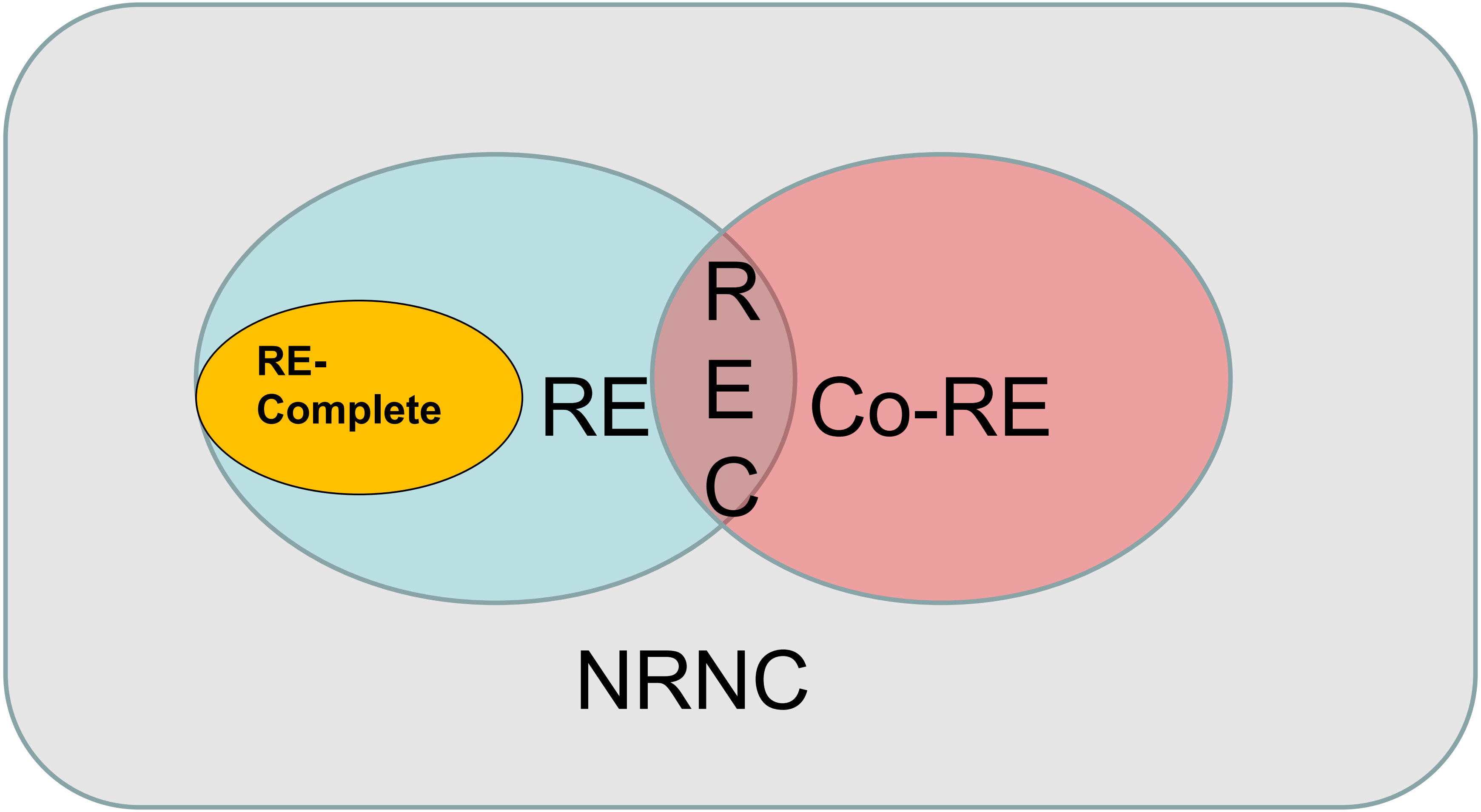
Gödel

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

Turing (Post, Church, Kleene)

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

UNIVERSE OF SETS



NonRE = (NRNC ∪ Co-RE) - REC

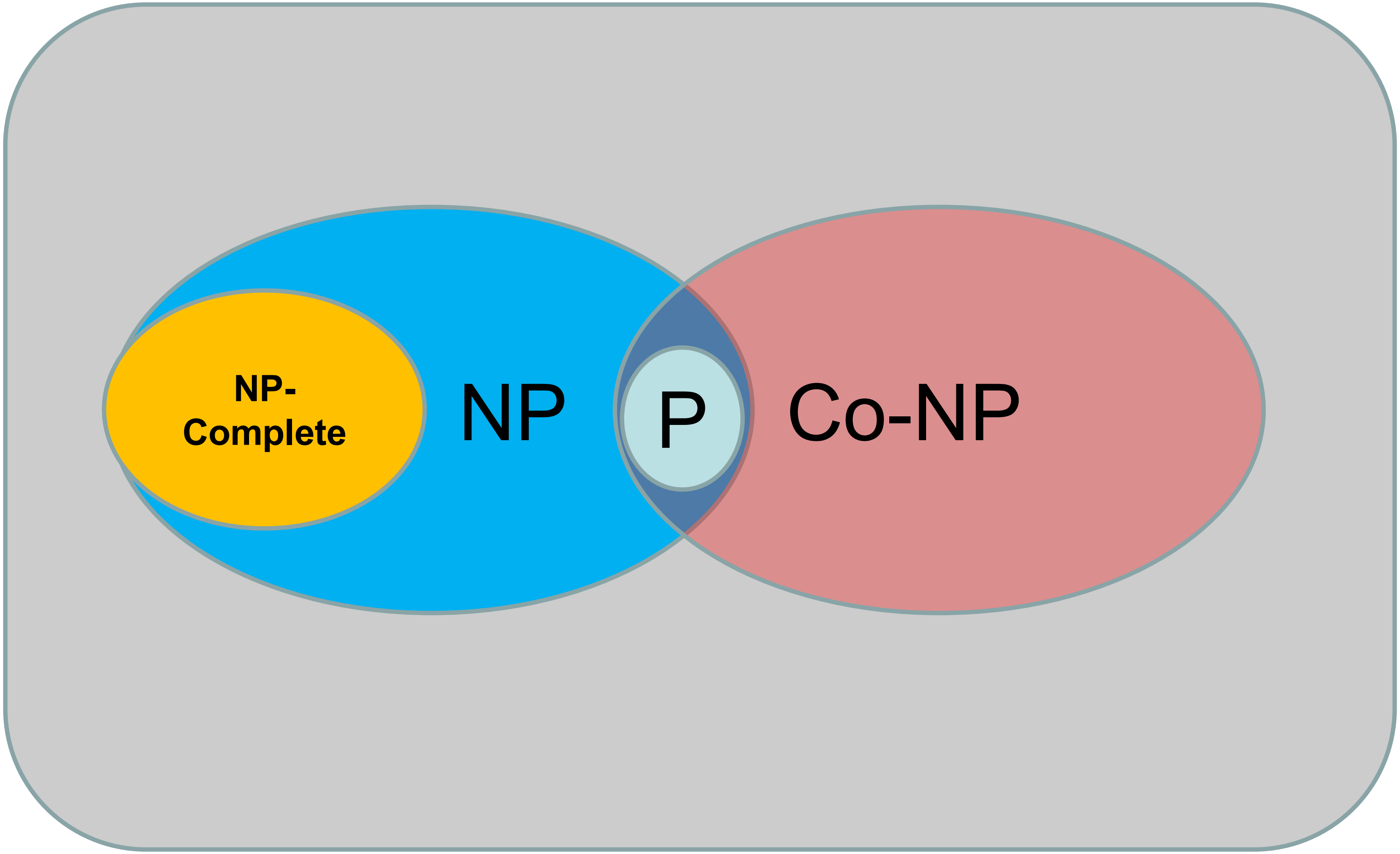
Complexity vs ..

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

P and NP

- P is the set (class) of problems solvable in polynomial time using a computer with a fixed number of processors.
- NP is the set of problems solvable in polynomial time using a finite but unbounded number of processors.
- Note: P vs NP also means deterministic versus non-deterministic polynomial time.
- Big question: Is P = NP?

UNIVERSE OF SETS



Regular Languages

Includes and Expands on
Chapter 1 of Sipser

Regular Languages # 1

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

Regular Languages # 2

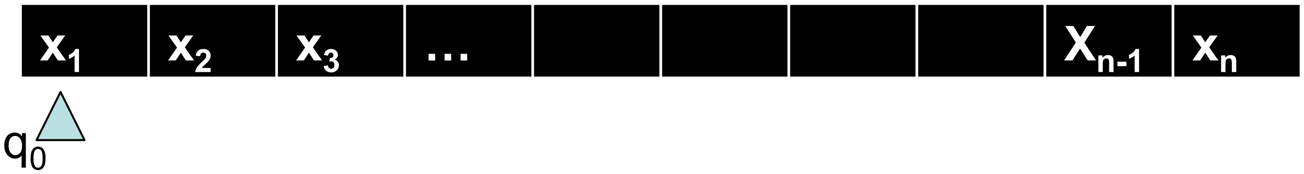
- Regular equations: REQs and FSAs.
- Myhill-Nerode Theorem: Right invariant equivalence relations. Specific relation for a language L. Proof and applications.
- Minimization: Why it's unique. Process of minimization. Analysis of cost of different approaches.
- Regular (right linear) grammars, regular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

Regular Languages # 3

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

Concrete Model of FSA

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



- 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 δ: $Q \times \Sigma \rightarrow Q$.
- 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 $F \subseteq Q$.

Finite State Automata

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

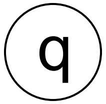
DFA Transitions

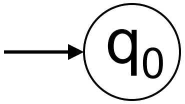
- Given a DFA, $A = (Q,\Sigma,\delta,q_0,F)$, we can definition the reflexive transitive closure of δ, $\delta^*:Q \times \Sigma^* \rightarrow Q$, by
  - $\delta^*(q,\lambda) = q$ where λ is the string of length 0
    - Note that text uses ϵ rather than λ as symbol for string of length zero
  - $\delta^*(q,ax)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

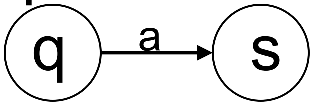
Regular Languages and DFAs

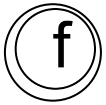
- Given a DFA, $A = (Q,\Sigma,\delta,q_0,F)$, we can define the language accepted by A as those strings that cause it to end up in a final state once it has consumed the entire string
- Formally, the language accepted by A is
  - $\{ w \mid \delta^*(q_0,w) \in F \}$
- We generally refer to this language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

State Diagram

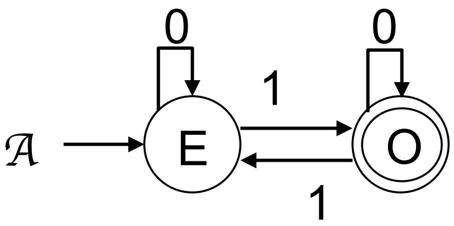
- A finite state automaton can be described by a state diagram, where
  - Each state is represented by a node labelled with that state, e.g., 

  - The state state has an arc entering it with no source, e.g., 

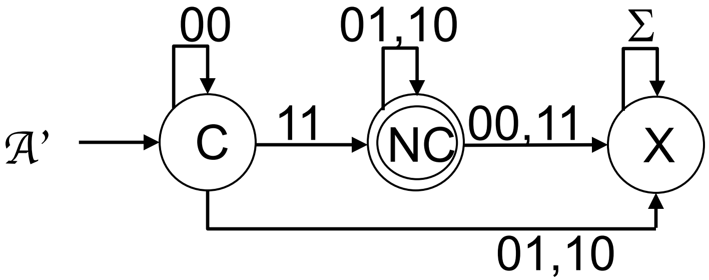
  - Each transition δ(q,a) = s is represented by a directed arc from node q to node s that is labelled with the letter a, e.g., 

  - Each final state has an extra circle around its node, e.g., 

Sample DFAs # 1

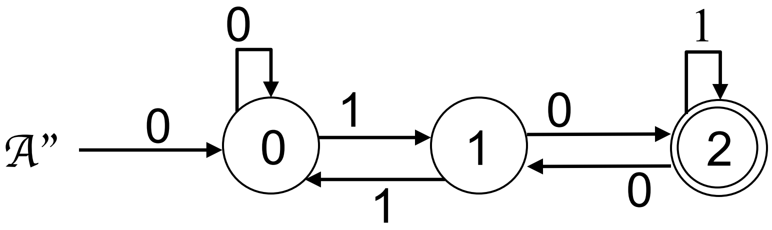


𝒜 = ({E,O}, {0,1}, δ, E, {O}), where δ is defined by above diagram.
L(𝒜) = { w | w is a binary string of odd parity }



𝒜' = ({C,NC,X}, {00,01,10,11}, δ', C, {NC}), where δ' is defined by above diagram.
L(𝒜') = { w | w 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 DFAs # 2



$\mathcal{A}$” = ({0,1,2}, {0,1}, δ, 0, {2}), where δ” is defined by above diagram.
L($\mathcal{A}$”) = { 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 |Σ| columns
- Rows are labelled with state names and columns with input letters
- The start state has some indicator, e.g., a greater than sign (>q) and each final state has some indicator, e.g., an underscore ($\underline{f}$)
- The entry in row q, column a, contains δ(q,a)
- In general we will use state diagrams, but transition tables are useful in some cases (state minimization)

FSAs and Applications

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

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
- $\Sigma^*\text{-}L(A_1)$ is recognized by $A_1^C = (Q_1,\Sigma,\delta_1,q_0,Q_1\text{-}F_1)$
- Define $A_3 = (Q_1 \times Q_2,\Sigma,\delta_3,<q_0,s_0>,F_3)$ where $\delta_3(<q,s>,a)= <\delta_1(q,a),\delta_2(s,a)>$, $q\in Q_1$, $s\in Q_2$, $a\in\Sigma$
  - $L(A_1) \cup L(A_2)$ 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 $F_3=F_1 \times (Q_2\text{-}F_2)$
  - $L(A_1) \oplus L(A_2)$ is recognized when $F_3=F_1 \times (Q_2\text{-}F_2) \cup (Q_1\text{-}F_1) \times F_2$

Complement of Regular Sets

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

NFA Transitions

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

NFA Languages

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

Finite State Diagram

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

Sample NFAs

- Done in class

Equivalence of DFA and NFA

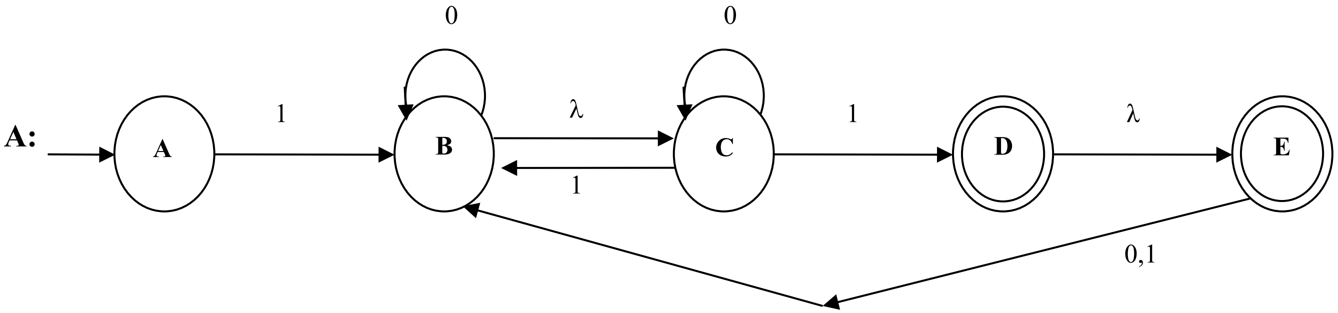
- Clearly every DFA is an NFA except that $\delta(q,a) = s$ becomes $\delta(q,a) = \{s\}$, so any language accepted by a DFA can be accepted by an NFA.
- The challenge is to show every language accepted by an NFA is accepted by an equivalent DFA. That is, if A is an NFA, then we can construct a DFA A',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

Constructing DFA from NFA

- Let A = (Q,Σ,δ,q_0,F) be an arbitrary NFA
- Let S be an arbitrary subset of Q.
  - Construct the sequence seq(S) to be a sequence that contains all elements of S in lexicographical order, using angle brackets to . That is, if S={q1, q3, q2} then seq(S)=<q1,q2,q3>. If S=Ø then seq(S)=<>
- Our goal is to create a DFA, A', whose state set contains 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 λ-Closure(q) = $\{ t \mid t \in \delta^*(q,\lambda) \}$
- We can extend this to $S \in P(Q)$ by
  λ-Closure(S) = $\{ t \mid t \in \delta^*(q,\lambda), q \in S \} = \{ t \mid t \in \lambda\text{-Closure}(q), q \in S \}$



State	A	B	C	D	E
λ-closure	{ A }	{ B , C }	{ C }	{ D, E }	{ E }

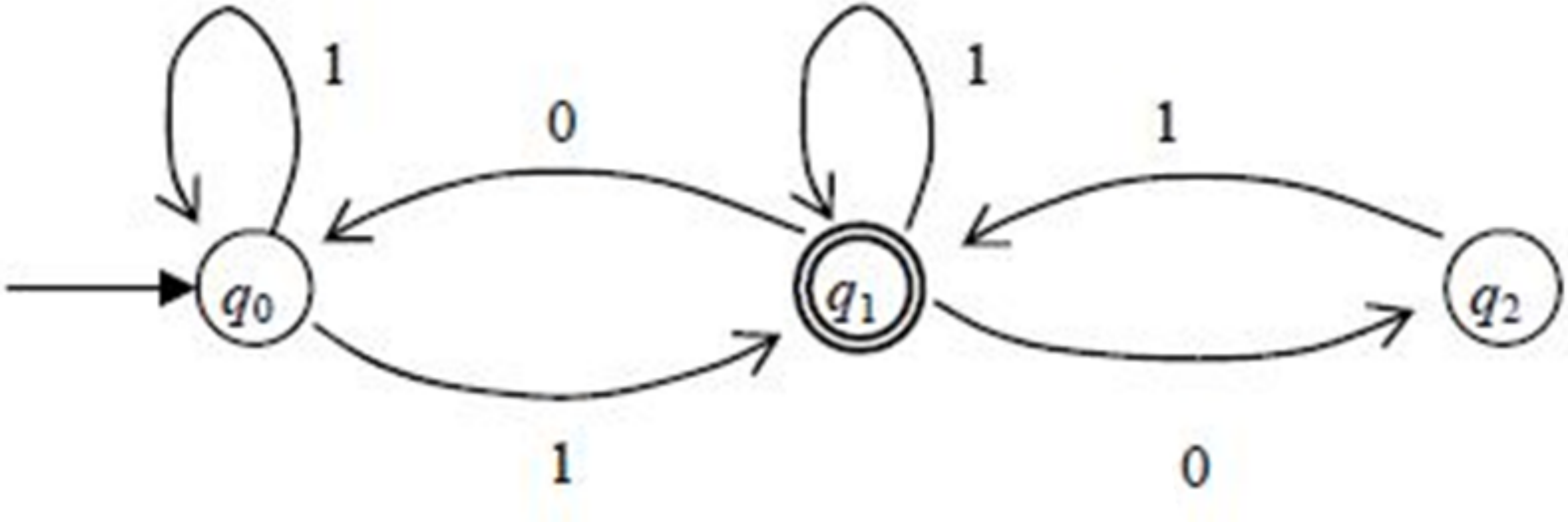
Details of DFA

- Let A = (Q,Σ,δ,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 λ-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≠λ 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.

Assignment # 3

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 seven. Thus, 111, 001110 and 010101 are in the language, but 101, 1001 and 11001 are not.
2. a.) Present a transition diagram with no lambda transitions for an NFA associated with the regular expression (011 + 0110 + 01 + 010)*.
   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. Hint: This DFA should have no more than six states.

Due: Thursday, September 21, 1:30PM (use Webcourses to turn in)
Extension due to Hurricane Irma!!

Regular Expressions

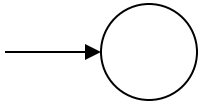
- Primitive:
  - Φ denotes {}
  - λ denotes {λ}
  - a where a is in Σ denotes {a}
- Closure:
  - If R and S are regular expressions then so are R ▪ S, R + S and R*, where
    - R ▪ S denotes RS = { xy | x is in R and y is in S }
    - R + S denotes R∪S = { x | x is in R or x is in S }
    - R* denotes R*
- Parentheses are used as needed

Regular Sets = Regular Languages

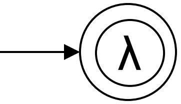
- Show every regular expression denotes a language recognized by a finite state automaton (can do deterministic or non-deterministic)
- Show every Finite State Automata recognizes a language denoted by a regular expression

Every Regular Set is a Regular Language

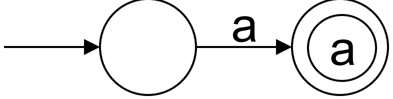
- Primitive:
  - Φ denotes {}



  - λ denotes {λ}



  - a where a is in Σ denotes {a}

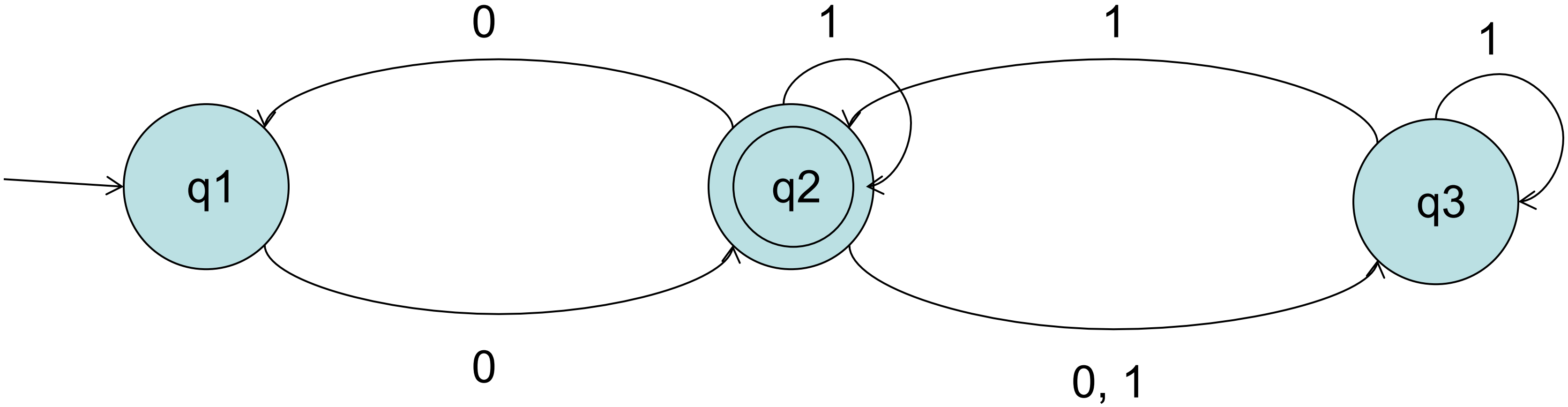


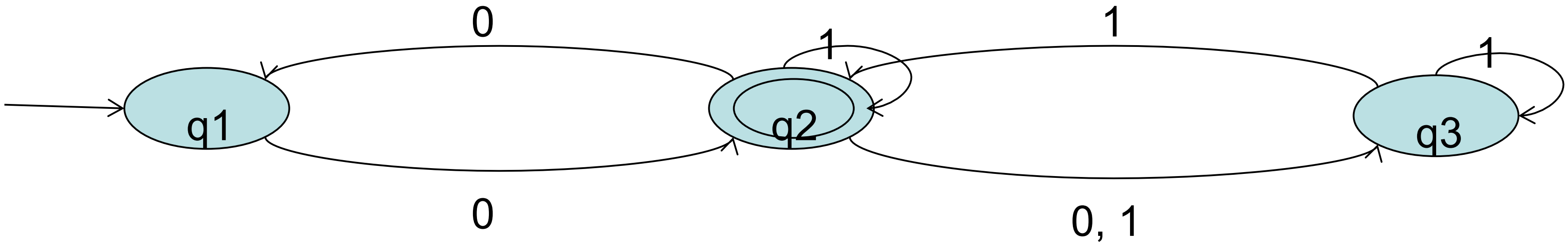
- Closure: (Assume that R’s and S’s states do not overlap)
  - R · S start with machine for R, add λ transitions from every final state of R’s recognizer to start state of S, making final state of S final states of new machine
  - R + S create new start state and add λ transitions from new state to start states of each of R and S, making union of R’s and S’s final states the new final states
  - R* add λ transitions from each final state of R back to its start state, keeping original start and final states (gets R^+) – FIX?

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

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

Convert to RE





- $R_{11}^{0}= \lambda$ $R_{12}^{0}= 0$ $R_{13}^{0}= \phi$
- $R_{21}^{0}= 0$ $R_{22}^{0}= \lambda + 1$ $R_{23}^{0}= 0 + 1$
- $R_{31}^{0}= \phi$ $R_{32}^{0}= 1$ $R_{33}^{0}= \lambda + 1$
- $R_{11}^{1}= \lambda$ $R_{12}^{1}= 0$ $R_{13}^{1}= \phi$
- $R_{21}^{1}= 0$ $R_{22}^{1}= \lambda + 1 + 00$ $R_{23}^{1}= 0 + 1$
- $R_{31}^{1} = \phi$ $R_{32}^{1}= 1$ $R_{33}^{1}= \lambda + 1$
- $R_{11}^{2}= \lambda + 01^*0$ $R_{12}^{2}= 0(1+00)^*$ $R_{13}^{2}= 0(1+00)^*(0+1)$
- $R_{21}^{2}= (1+00)^*0$ $R_{22}^{2}= (1+00)^*$ $R_{23}^{2}= (1+00)^*(0+1)$
- $R_{31}^{2}= 1(1+00)^*0$ $R_{32}^{2}= 1(1+00)^*$ $R_{33}^{2}= \lambda+1+1(1+00)^*(0+1)$
- $L = R_{12}^{3}=$
  $0(1+00)^* + 0(1+00)^*(0+1)\ (1+1(1+00)^*(0+1))^*\ 1(1+00)^*$

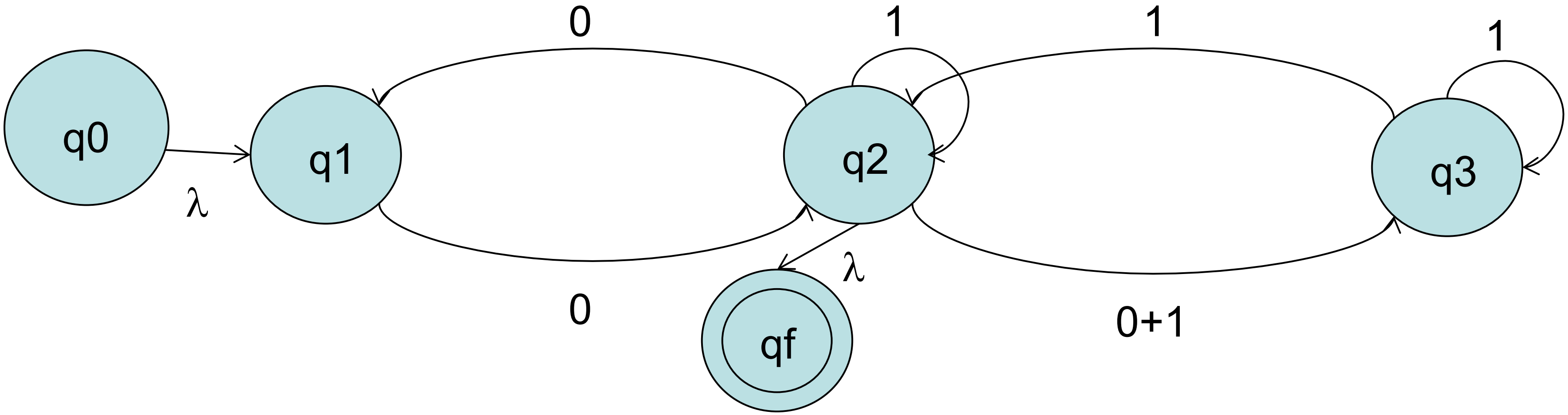
State Ripping Concept

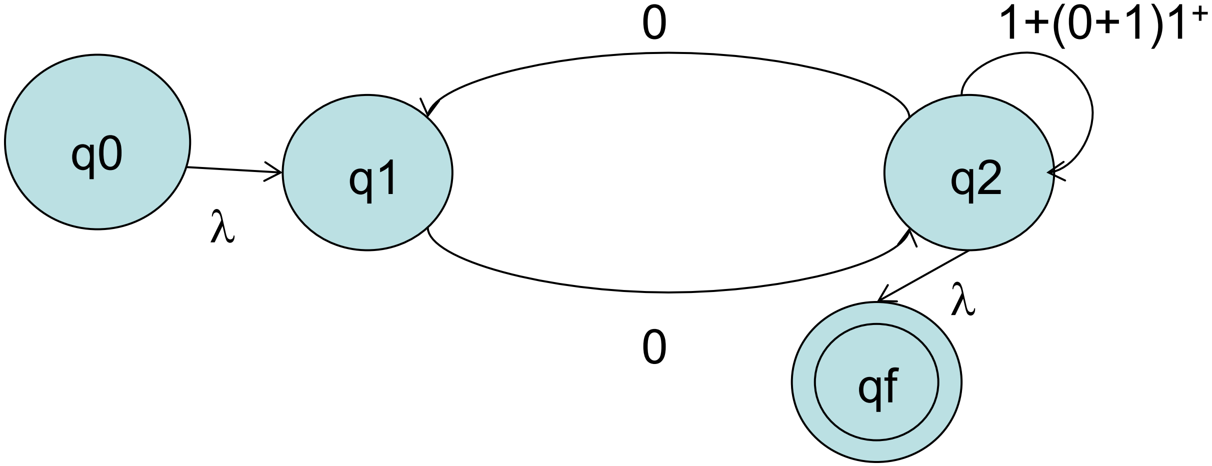
- This is similar to generalized automata approach but with fewer arcs than text. It actually gets some of its motivation from R_{ij}^k approach as well
- Add a new start state and add a λ–transition to existing start state
- Add a new final state q_f and insert λ–transitions from all existing final states to the new one; make the old final states non-final
- Leaving the start and final states, successively pick states to remove
- For each state to be removed, change the arcs of every pair of externally entering and exiting arcs to reflect the regular expression that describes all strings that could result 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

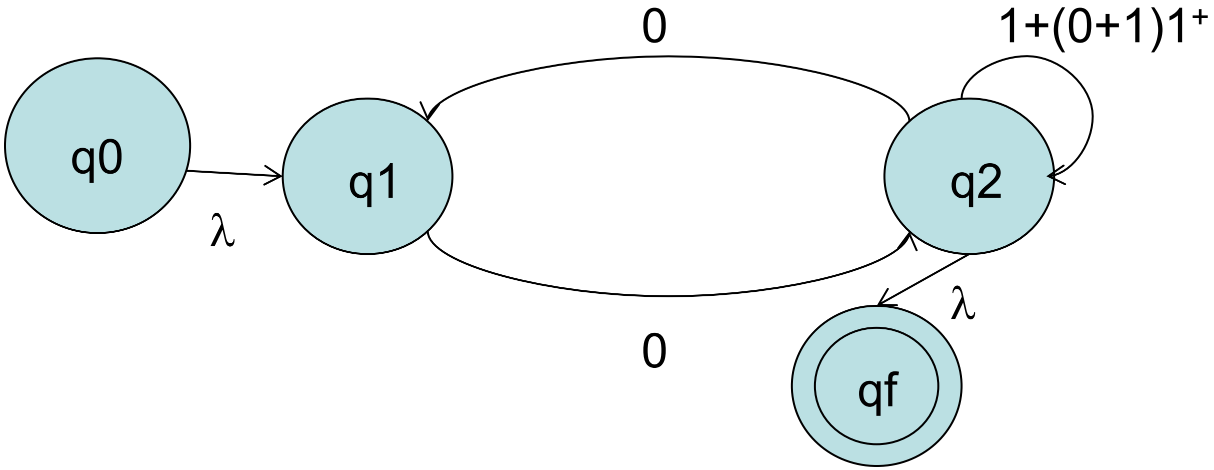
- Let B be the node to be removed
- Let e1 be the regular expression on the arc from some node A to some node B (A≠B); e2 be the expression from B back to B (or λ if there is no recursive arc); e3 be the expression on the arc from B to some other node C (C ≠B but C could be A); e4 be the expression from 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

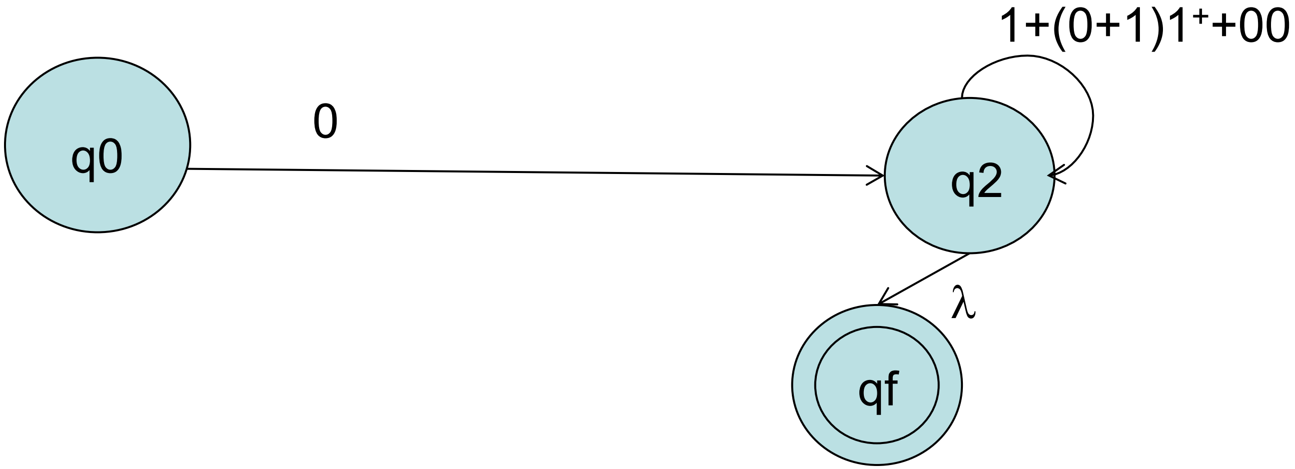
Use Ripping; Rip q3



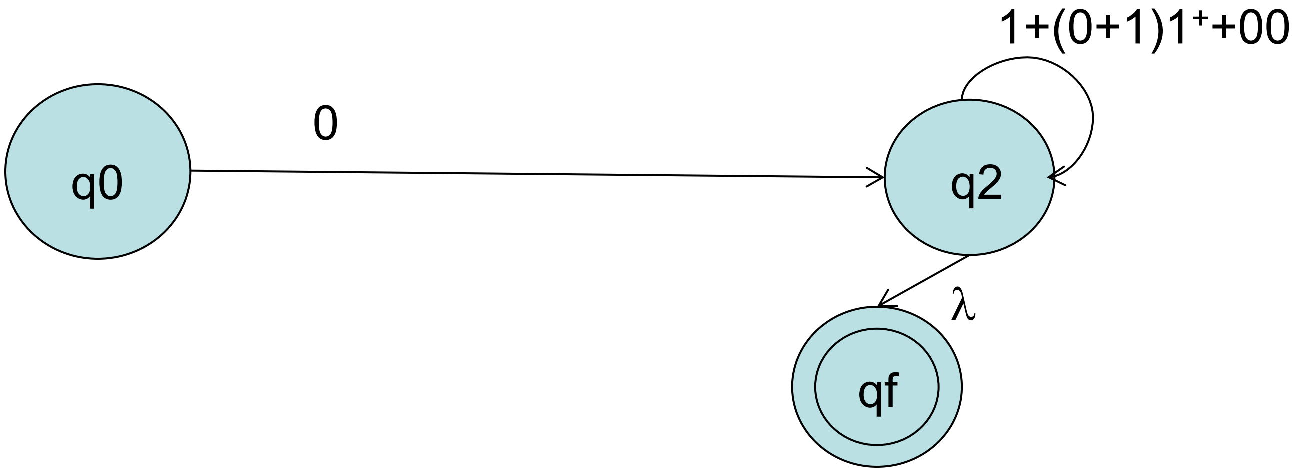


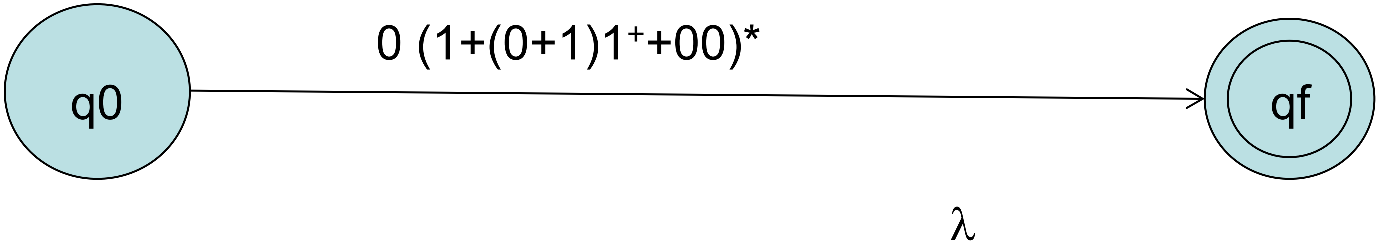
Use Ripping; Rip q1





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

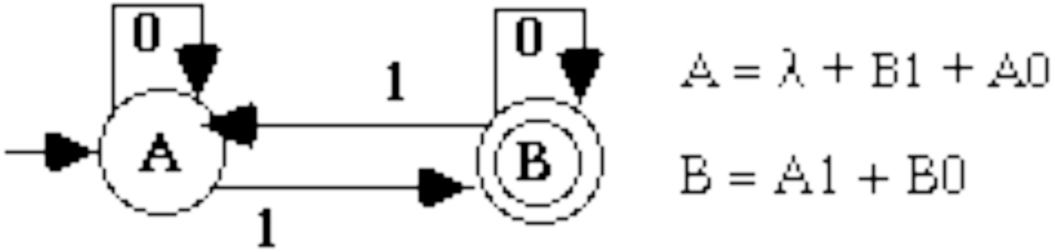
- We first show that QP* is contained in R. By definition, R = Q + RP.
- To see if QP* is a solution, we insert it as the value of R in Q + RP and see if the equation balances
- $R = Q + QP^*P = Q(\lambda+P^*P) = QP^*$
- Hence QP* is a solution, but not necessarily the only solution.

Uniqueness of Solution

- To prove uniqueness, we show that R is contained in QP*.
- By definition, $R = Q+RP = Q+(Q+RP)P$
- $= Q+QP+RP^2 = Q+QP+(Q+RP)P^2$
- $= Q+QP+QP^2+RP^3$
- ...
- $= Q(\lambda+P+P^2+ \ldots +P^i)+RP^{i+1}$, for all i>=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*.

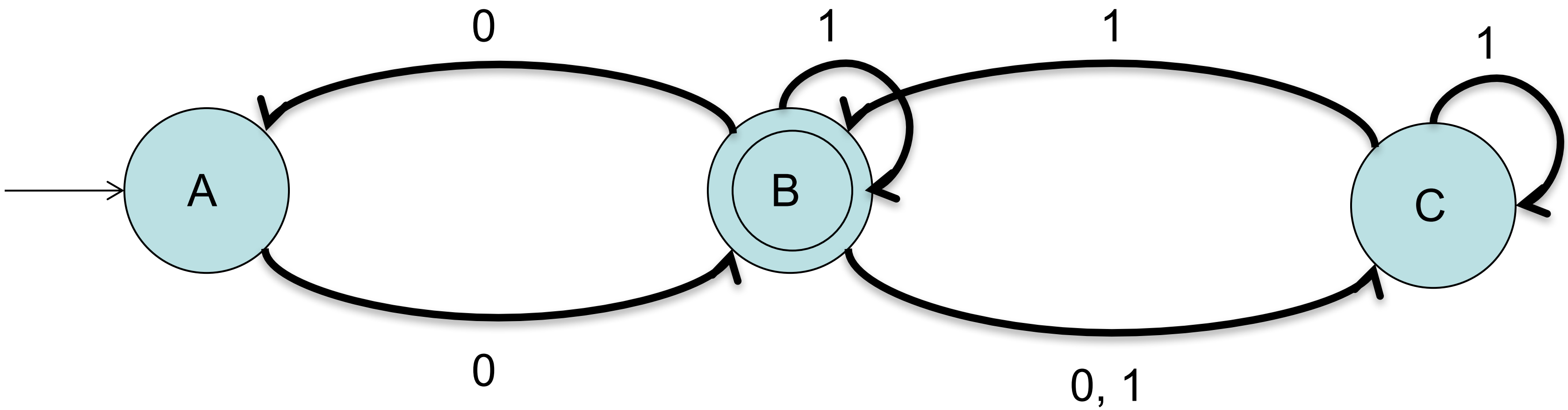
Example

- 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=λ+B1; P=0
  A = QP* = (λ+B1)0*
  = B10* + 0*



- B = B10*1 + B0 + 0*1
  For B, Q=0*1; P= B10*1 + B0 = B(10*1 + 0)
- and therefore
- B = 0*1(10*1 + 0)*
- Note: This technique fails if there are lambda transitions.

Using Regular Equations



A = λ + 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

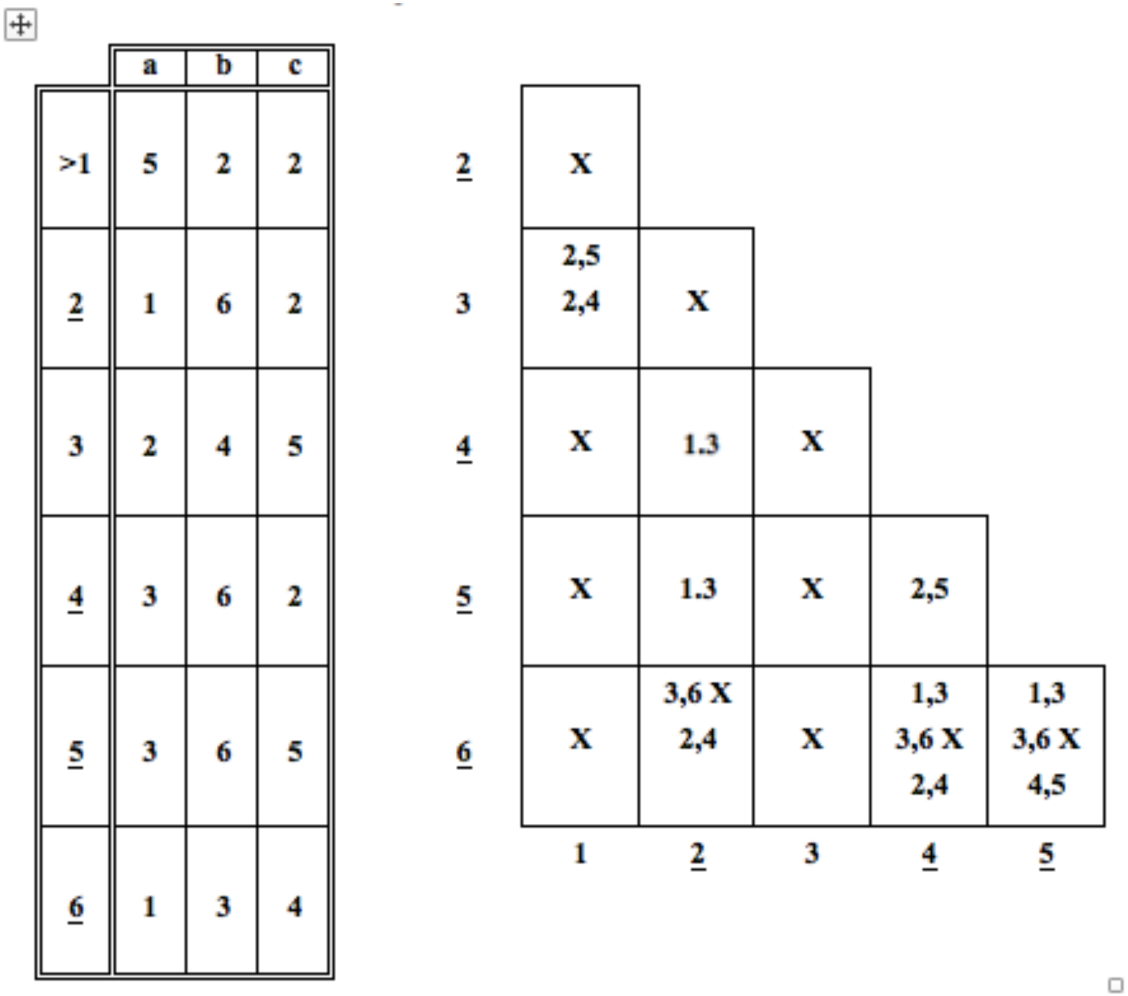
- For each of the DFAs you created for the previous page, use ripping of states and then regular equations to compute the associated regular expression. Note: You obviously ought to get expressions that are equivalent to the initial expressions.

State Minimization

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

Sample Minimization

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



	a	b	c
>1	5	2	2
2	1	6	2
3	2	4	5
4	3	6	2
5	3	6	5
6	1	3	4

2	X				
3	2,5 2,4	X			
4	X	1.3	X		
5	X	1.3	X	2,5	
6	X	3,6 X 2,4	X	1,3 3,6 X 2,4	1,3 3,6 X 4,5
	1	2	3	4	5

Reversal of Regular Sets

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

Substitution

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

Quotient with Regular Sets

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

Quotient Again

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

Applying Meta Approach

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

Making Life Easy

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

Reachable and Reaching

- Reachable*from*(q) = { p | $\exists$ w $\ni$ δ(q,w)=p }
  - Just do depth first search from q, marking all reachable states. Works for NFA as well.
- Reaching*to*(q) = { p | $\exists$ w $\ni$ δ(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$\in$L and no proper prefix of w is in L } = { w | w$\in$L and if w=xy, x$\in\Sigma^*$, y$\in\Sigma^+$ then x$\notin$L}
- Max(L) = { w | w$\in$L and w is not the proper prefix of any word in L } = { w | w$\in$L and if y$\in\Sigma^+$ then wy$\notin$L }
- Examples:
  - Min(0(0+1)*) = {0}
  - Max(0(0+1)*) = {}
  - Min(01 + 0 + 10) = {0,10}
  - Max(01 + 0 + 10) = {01,10}
  - Min({$a^ib^jc^k$ | i ≤ k or j ≤ k}) = {$a^ib^jc^k$ | | i,j ≥0, k = min(i, j)}
  - Max({$a^ib^jc^k$ | i ≤ k or j ≤ k}) = {} because k has no bound
  - Min({$a^ib^jc^k$ | i ≥ k or j ≥ k}) = {Λ}
  - Max({$a^ib^jc^k$ | i ≥ k or j ≥ k}) = {$a^ib^jc^k$ | | i,j ≥0, k = 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,q_0,F) is a DFA with no state unreachable from q_0
- Define $A_{min} = (Q \cup \{dead\},\Sigma,\delta_{min},q_0,F)$, where for $a \in \Sigma$ $\delta_{min}(q,a) = \delta(q,a)$, if $q \in Q-F$; $\delta_{min}(q,a) = dead$, if $q \in F$; $\delta_{min}(dead,a) = dead$

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

Therefore, Regular Languages are closed under Min.

Regular Closed under Max

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

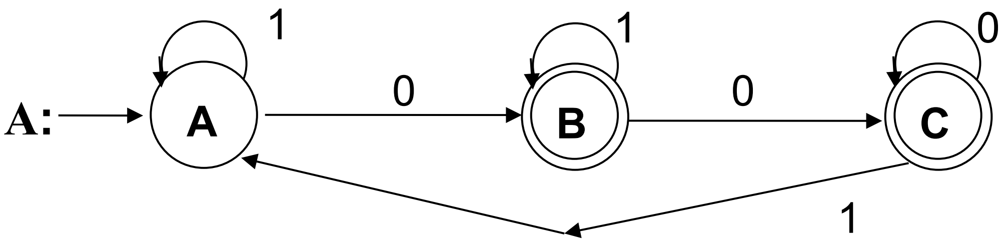
The inductive definition of Reachable*from*$^+$ is:

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

Therefore, Regular Languages are closed under Max.

Assignment # 4.1

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



Due: Tuesday, September 26, 1:30PM (Use Webcourses to turn in)

Assignment # 4.2

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

	a	b	c
>1	2	3	5
2	5	4	4
3	2	4	5
4	6	4	2
5	5	2	4
6	5	4	2

2					
3					
4					
5					
6					
	>1	2	3	4	5

Construct and write down your new, equivalent automaton!!

Due: Tuesday, September 26, 1:30PM (use Webcourses to turn in)

Pumping Lemma Concept

- Let A = (Q,Σ,δ,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 $i \geq 0$, $xy^iz \in L$
- 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 $x = v_1\ldots v_j$, $y = v_{j+1}\ldots v_k$, 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 $|xy| \le N$, $|y| > 0$, and for all $i \ge 0$, $xy^i z \in L$
  - We CANNOT choose split, but P.L. is constrained by N
- Our turn: Choose i = 0.
  - We have the power here
- P.L: $a^{N-|y|} b^N \in L$; just a consequence of P.L.
- Our turn: $a^{N-|y|} b^N \notin L$; just a consequence of L's structure
- CONTRADICTION, so L is NOT regular

xwx is not Regular (PL)

- **L = { x w x | 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 = $a^N baa^N b$
- PL: Since s $\in$ L and |s| ≥ N, s can be split into 3 pieces, s = xyz, such that |xy| ≤ N and |y| > 0 and $\forall$ i ≥ 0 $xy^iz \in$ L
- YOU: Choose i = 2
- PL: xy^2z = xyyz $\in$ L (could also use i = 0)
- Thus, $a^{N + |y|}baa^Nb$ would be in L, but this is not so since N+|y| ≠ N
- We have arrived at a contradiction.
- Therefore L is not Regular.

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

- **L = {$a^{Fib(k)}$ | k>0} :**
- Assume that L is regular
- Let N be the positive integer given by the Pumping Lemma
- Let *s* be a string **s = $a^{Fib(N+3)} \in$ L**
- Since *s* $\in$ L and |s| ≥ N (Fib(N+3)>N in all cases; actually Fib(N+2)>N as well), s is split by PL into xyz, where |xy| ≤ N and |y| > 0 and for all i ≥ 0, $xy^iz \in$ L
- We choose i = 2; by PL: xy^2z = xyyz$\in$ L
- Thus, $a^{Fib(N+3)+|y|}$ 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
  - $\{ 0^m 1^{2n} \mid m \leq n \}$
  - $\{ ww^R \mid w \in \{a,b\}^+ \}$
  - $\{ 1^{n^2} \mid n > 0 \}$
  - $\{ ww \mid w \in \{a,b\}^+ \}$

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 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 L iff $\forall z$ xz R_A yz. Thus, R_A is right invariant; its index is |Q| which is finite; and $L(A) = \cup_{\delta^*(x)\in F}[x]_{RA}$, where $[x]_{RA}$ refers to the equivalence class containing the string x.

Myhill-Nerode 2 $\Rightarrow$ 3

2. Assume L is the union of some of the classes of a right invariant equivalence relation, R, of finite index.
3. Since x R y iff $\forall$z [xz R yz], R is right invariant and L is the union of some of the equivalence classes, then
   x R y $\Rightarrow$ $\forall$z [xz $\in$ L iff yz $\in$ L] $\Rightarrow$ x R_L y.
   This means that the index of R_L is less than or equal to that of R and so is finite. Note than 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
   $Q = \{\ [x]_{RL} \mid x \in \Sigma^*\ \}$
   $\delta([x]_{RL},a) = [xa]_{RL}$
   $q1 = [\lambda]$
   $F = \{\ [x]_{RL} \mid x \in L\ \}$

   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 $a^j b b^{i-1} \notin L$ and so $[a^i b]$ is not related to $[a^j b]$ under R_L and thus $[a^i b] \neq [a^j b]$.
- This means that R_L has infinite index.
- Therefore L is not regular.

xwx is not Regular (MN)

- **L = { x a x | x$\in${a,b}+} :**
- We consider the right invariant equivalence class [a^ib], i>0.
- It's clear that a^ibaa^{i}b is in the language, but a^kbaa^{i}b 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.

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

- **L = $\{a^{Fib(k)} \mid k>0\}$:**
- We consider the collection of right invariant equivalence classes $[a^{Fib(j)}]$, j > 2.
- It's clear that $a^{Fib(j)}a^{Fib(j+1)}$ is in the language, but $a^{Fib(k)}a^{Fib(j+1)}$ is not when k>2 and k≠j and k≠j+2
- This shows that there is a separate equivalence class $[a^{Fib(j)}]$ 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 Lnaguges
  - Myhill-Nerode Theorem

Finite State Transducers

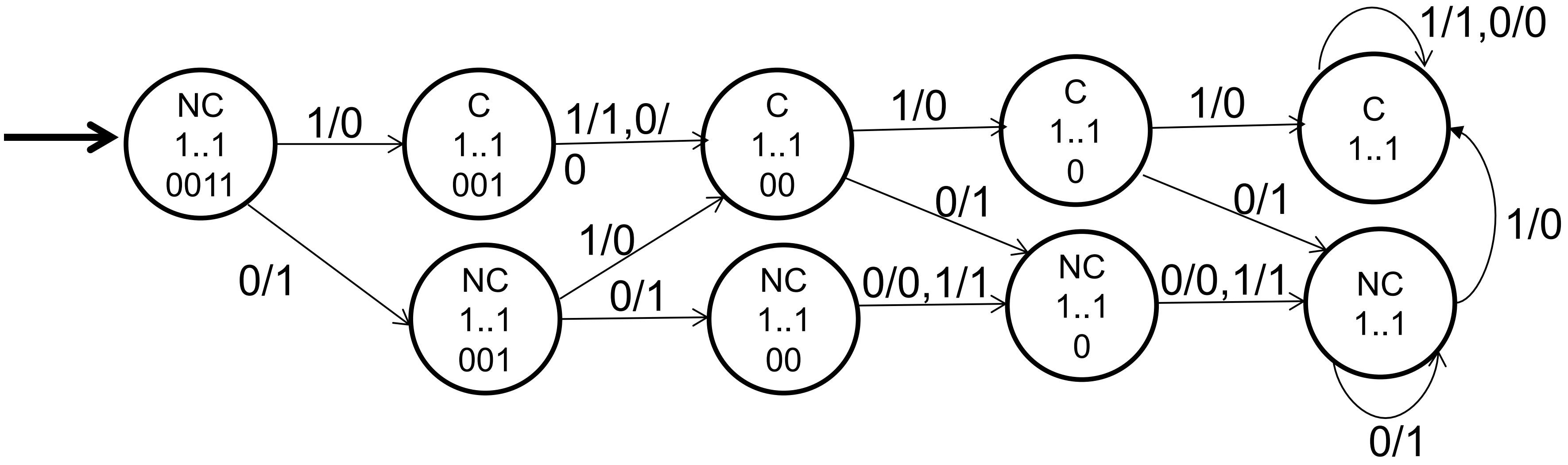
- A transducer is a machine with output
- Mealy Model
  - $M = (Q, \Sigma, \Gamma, \delta, \gamma, q_0)$

    Γ is the finite output alphabet

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

Sample Mealy Model

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



Finite State Transducers

- Moore Model
  - $M = (Q, \Sigma, \Gamma, \delta, \gamma, q_0)$

    Γ is the finite output alphabet

    $\gamma: Q \to \Gamma$ 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.

Assignment # 5

1. For each of the following, prove it is not regular by using the Pumping Lemma or Myhill-Nerode. You must do at least one of these using the Pumping Lemma and at least one using Myhill-Nerode.

a. **L = { x#y | x, y$\in$ {0,1}$^+$ and y is the twos complement of x }**

b. **L = { $a^i b^j c^k$ | i>k or j>k or k>i }**

c. **L = { x w x | x, w $\in$ {a,b}$^+$ and |x| = |w| }**

2. Write a regular (right linear) grammar that generates **L = { w | w $\in$ {0,1}$^+$** and **w** interpreted as a binary number has a remainder of 3 or 4 when divided by 6 } .

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 **101** to **x** (assumes all numbers are positive, including results). Assume that **x** is read starting with its least significant digit.
Examples: **0010** → **0111; 0101** → **1010; 0001** → **0110; 0111** → **1100**

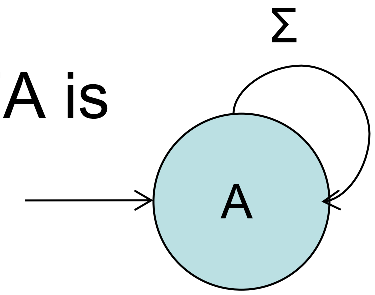
Due: Friday, September 29, 11:59PM (use Webcourses to turn in)

Decision and Closure Properties

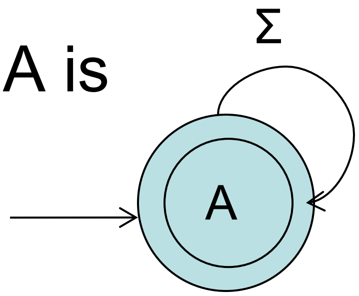
Regular Languages

Decidable Properties

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



- L = Σ*: 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

Midterm#1 Topics.1

- Properties of sets, sequences, relations, functions and graphs
  - Basic notions
  - Proof techniques
- Computability, complexity, languages
  - Basic notions

Midterm#1 Topics.2

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

Midterm#1 Topics.3

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

Midterm#1 Topics.4

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

Midterm#1 Topics.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 = Ø
  - L = Σ*
  - Finiteness / Infiniteness
  - Equivalence

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

Derivations

- $x \Rightarrow y$ reads as x derives y iff
  - $x = \gamma\alpha\delta$, $y = \gamma\beta\delta$ and $\alpha \to \beta$
- $\Rightarrow^*$ 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:

$A \rightarrow a$, $A \in V$, $a \in \Sigma^*$

$A \rightarrow \lambda$, $A \in V$, $a \in \Sigma^*$

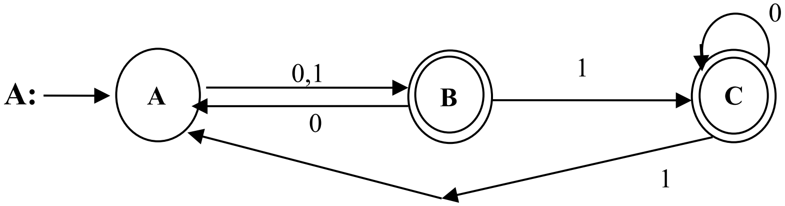
$A \rightarrow aB$, $A, B \in V$, $a \in \Sigma^*$

DFA to Regular Grammar

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

Example of DFA to Grammar

- DFA



- Grammar

A → 0 B | 1 B

B → 0 A | 1 C | λ

C → 0 C | 1 A | λ

Regular Grammar to NFA

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

Example of Grammar to NFA

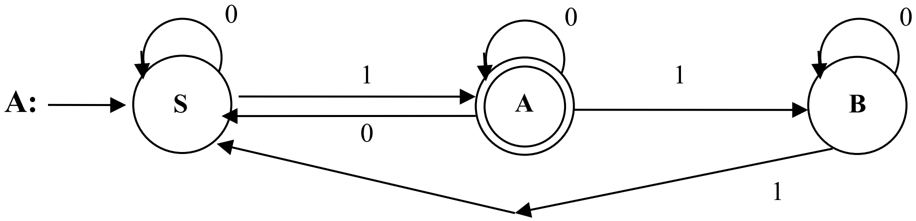
- Grammar

S → 0 S | 1 A

A → 0 S | 0 A | 1 B | λ

B → 1 S | 0 B

- DFA



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}, Σ, R, S), where R is:
  - S → aT
  - T → 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

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

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

The right hand side is a string from the combined alphabets

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

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

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

Interesting Sample CFG

Example of a grammar for a small language:

G = ({<program>, <stmt-list>, <stmt>, <expression>},
{begin, end, ident, ;, =, +, -}, R, <program>) where R is

<program> → begin <stmt-list> end
<stmt-list> → <stmt> | <stmt> ; <stmt-list>
<stmt> → ident = <expression>
<expression> → ident + ident | ident - ident | ident

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

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

Derivation

A sentence generation is called a derivation.

Grammar for a simple assignment statement:

R1	**<assgn>**	**→**	**<id> := <expr>**
R2	**<id>**	**→**	**a | b | c**
R3	**<expr>**	**→**	**<id> + <expr>**
R4		**|**	**<id> * <expr>**
R5		**|**	**(<expr>)**
R6		**|**	**<id>**

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

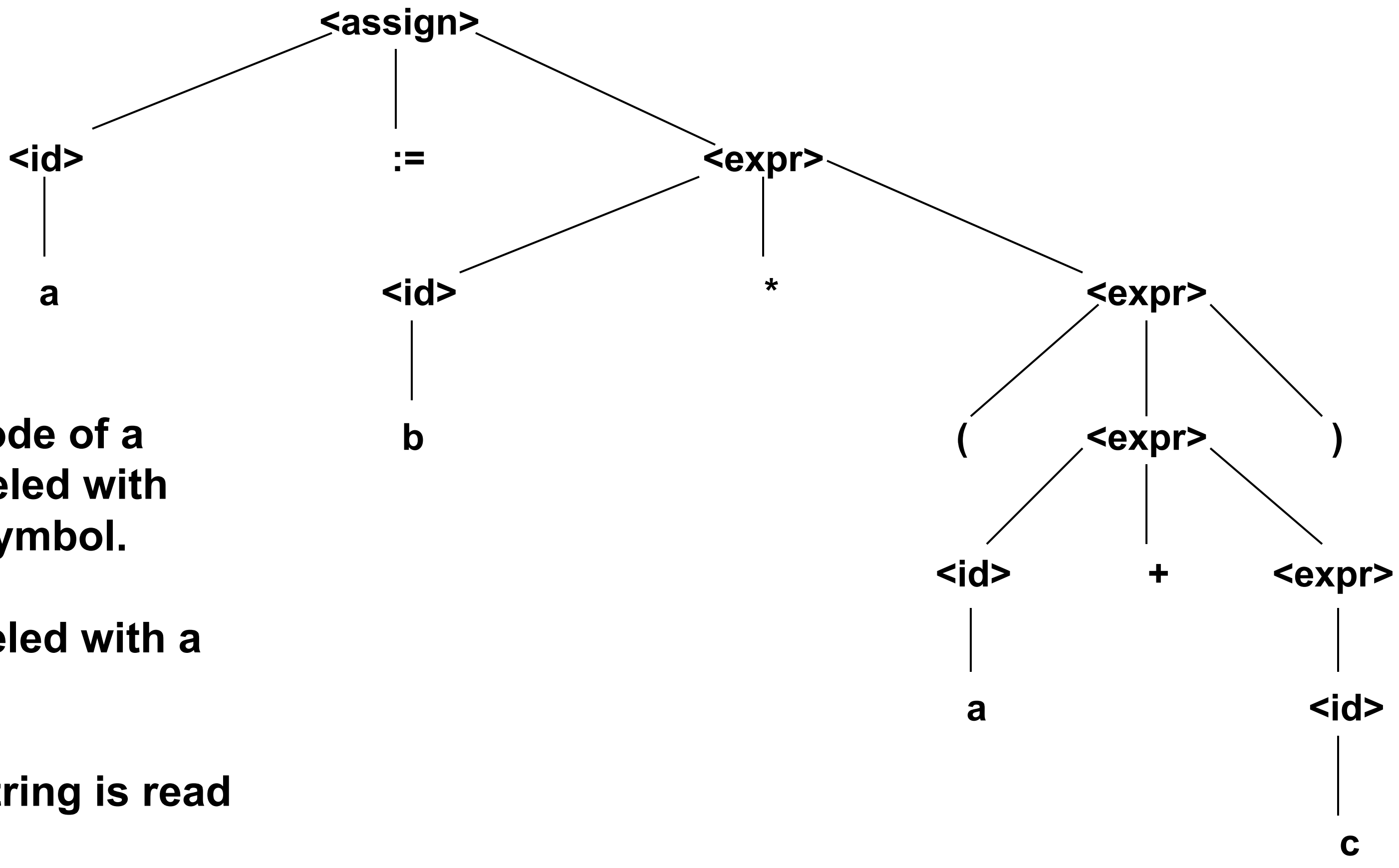
The statement a := b * (a + c)
Is generated by the leftmost derivation:

<assgn>	**⇒ <id> := <expr>**	**R1**
	⇒ a := <expr>	**R2**
	⇒ a := <id> * <expr>	**R4**
	⇒ a := b * <expr>	**R2**
	⇒ a := b * (<expr>)	**R5**
	⇒ a := b * (<id> + <expr>)	**R3**
	⇒ a := b * (a + <expr>)	**R2**
	⇒ a := b * (a + <id>)	**R6**
	⇒ a := b * (a + c)	**R2**

Parse Trees

<u>A parse tree is a graphical representation of a derivation</u>

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



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

Every leaf is labeled with a terminal symbol.

The generated string is read left to right

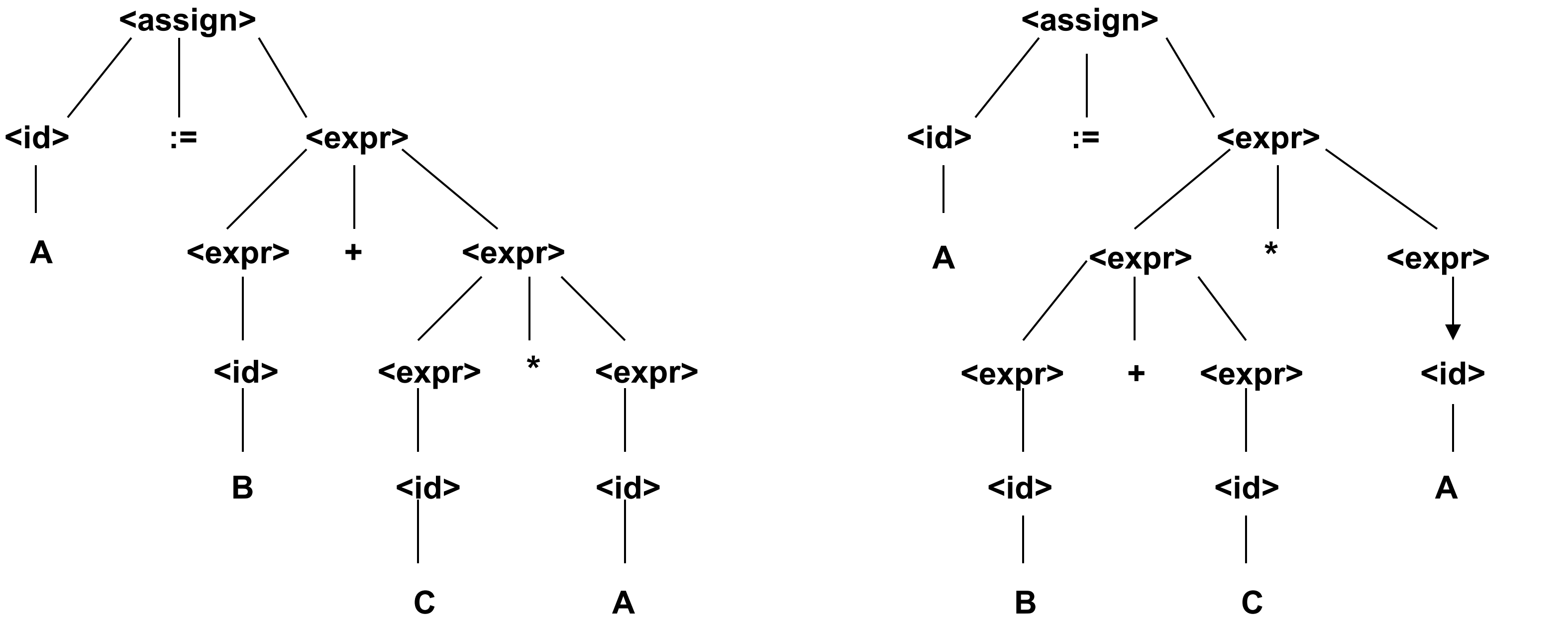
Ambiguity

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

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

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

Ambiguous Parse



This grammar generates two parse trees for the same expression.

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

Precedence

Operator precedence:

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

An unambiguous grammar for expressions:

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

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

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

Left (right)most Derivations

Leftmost derivation:

<assgn> → <id> := <expr>
→ a := <expr>
→ a := <expr> + <term>
→ a := <term> + <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:

<assgn> ⇒ <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:

<expr> → <expr> <op> <expr> | id | int | (<expr>)
<op>   → + | - | * | /

Can be rewritten as:

<expr> → <term> | <expr> + <term> | <expr> - <term>
<term> → <factor> | <term> * <factor> | <term> / <factor>.
<factor> → id | int | (<expr>)

Parsing Problem

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

For correct strings:

Sentence + grammar → parse tree

For a compiler, a sentence is a program:

Program + grammar → 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: Expr → Expr + Term | Term
Term → Term * Factor | Factor
Factor → (Expr) | Int

Fix: Expr → Term ExprRest
ExprRest → + Term ExprRest | λ
Term → Factor TermRest
TermRest → * Factor TermRest | λ
Factor → (Expr) | Int

Bottom Up vs Top Down

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

Chomsky Normal Form

- Each rule of a CFG is constrained to be of one of the three forms:
  $A \rightarrow a$, $A \in V$, $a \in \Sigma$
  $A \rightarrow BC$, $A,B,C \in V$
- If the language contains λ then we allow
  $S \rightarrow \lambda$
  and constrain all rules of form to be
  $A \rightarrow BC$, $A \in V$, $B,C \in V\text{-}\{S\}$

Nullable Symbols

- Let G = (V, Σ, R, S) be an arbitrary CFG
- Compute the set Nullable(G) = $\{A \mid A \Rightarrow^* \lambda\}$
- Nullable(G) is computed as follows

```
Nullable(G) ⊇ { A | A → λ }
Repeat
    Nullable(G) ⊇ { B | B → α and α ∈ Nullable* }
until no new symbols are added
```

Removal of λ-Rules

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

Chains (Unit Rules)

- Let G = (V, Σ, R, S) be an arbitrary CFG that has had its λ-rules removed
- For $A \in V$, $\mathrm{Chain}(A) = \{ B \mid A \Rightarrow^* B, B \in V \}$
- Chain(A) is computed as follows

  $\mathrm{Chain}(A) \supseteq \{ A \}$

  Repeat

  $\quad \mathrm{Chain}(A) \supseteq \{ C \mid B \rightarrow C \text{ and } B \in \mathrm{Chain}(A) \}$

  until no new symbols are added

Removal of Unit-Rules

- Let $G = (V, \Sigma, R, S)$ be an arbitrary CFG that has had its λ-rules removed, except perhaps from start symbol
- Compute 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 $B \in$ Chain(A)
  Note: $A \in$ Chain(A) so all its non unit-rules are included

Non-Productive Symbols

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

Unreachable Symbols

- Let G = (V, Σ, R, S) be an arbitrary CFG that has had its λ-rules, unit-rules and non-productive symbols removed
- Unreachable symbols are ones that are inaccessible from start symbol
- We compute the complement (Useful)
- Useful(G) is computed by

  Useful(G) $\supseteq$ { S }

  Repeat

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

  until no new symbols are added
- Keep only those rules that involve useful symbols
- If no rules remain, grammar generates nothing

Reduced CFG

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

CFG to CNF

- Let G = (V, Σ, R, S) be arbitrary reduced CFG
- Define G'=(V∪{<a>|a∈Σ}, Σ, R, S)
- Add the rules <a> → a, for all a∈Σ
- For any rule, A → α, |α| > 1, change each terminal symbol, a, in α to the non-terminal <a>
- Now, for each rule A → BCα, |α| > 0, introduce the new non-terminal B<Cα>, and replace the rule A → BCα with the rule A → B<Cα> and add the rule <Cα> → Cα
- Iteratively apply the above step until all rules are in CNF

Example of CNF Conversion

Starting Grammars

- L = { a^i b^j c^k | i=j or j=k }
- G = ({S,A,<B=C>,C,<A=B>}, {a,b}, R, S)
- R:
  - S → A | C
  - A → a A | <B=C>
  - <B=C> → b <B=C> c | λ
  - C → C c | <A=B>
  - <A=B> → a <A=B> b | λ

Remove Null Rules

- Nullable = {<B=C>, <A=B>, A, C, S}
  - S’ → S | λ // S’ added to V
  - 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

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’ → λ | aA | a | b<B=C>c | bc | Cc | c | a<A=B>b | ab
  - S → aA | a | b<B=C>c | bc | Cc | c | a<A=B>b | ab
  - A → aA | a | b<B=C>c | bc
  - <B=C> → b<B=C>c | bc
  - C → Cc | c | a<A=B>b | ab
  - <A=B> → 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’ → λ | <a>A | a | <b><<B=C><c>> | <b><c> | C<c> | c | <a><<A=B><b>> | <a><b>
- A → <a>A | a |<b><<B=C><c>> | <b><c>
- <B=C> → <b><<B=C><c>> | <b><c>
- C → C<c> | c | <a><<A=B><b>> | <a><b>
- <A=B> → <a> <<A=B><b>> | <a><b>
- <<B=C><c>> → <B=C><c>
- <<A=B><b>> → <A=B><b>
- <a> → a
- <b> → b
- <c> → 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 →	BC	where B and C are non-terminals
A →	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] = 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 $R_A$ -> $R_B$ $R_C$
                    If P[j,k,B] and P[j+k,i-k,C] then set P[j,i,A] = 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
```

CKY Parser

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

S → AB | BA

A → CD | a

B → CE | b

C → a | b

D → AC

E → BC

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

2^{nd} CKY Example

E → E F | M E | P E | a
F → M F | P F | M E | P E
P → +
M → –

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

Assignment # 6

1. Write a CFG for the following languages:
   L = { $a^ib^jc^kd^m$ | i = j+k or j = k+m or i = m }

2. Convert the following grammar to a CNF equivalent grammar. Show all steps.
   G = ({ S, E , F, A } , { a , (,) , -, +, ; } , R, S), where **R** is:
   S → E; S | E
   E → E – F | E + F | F
   F → aA | (E)
   A → aA | λ

3. Present the **CKY** recognition matrix for the string **a a a b b b** assuming the Chomsky Normal Form grammar **G = ({ S,T,U,A,B} , { a,b} , R, S)**, where **R** is specified by the rules
   S -> S T | U A | a
   T -> B U | b
   U -> A S
   A -> a
   B -> b

Due: Thursday 10/22, 1:30PM (use Webcourses to turn in)

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.

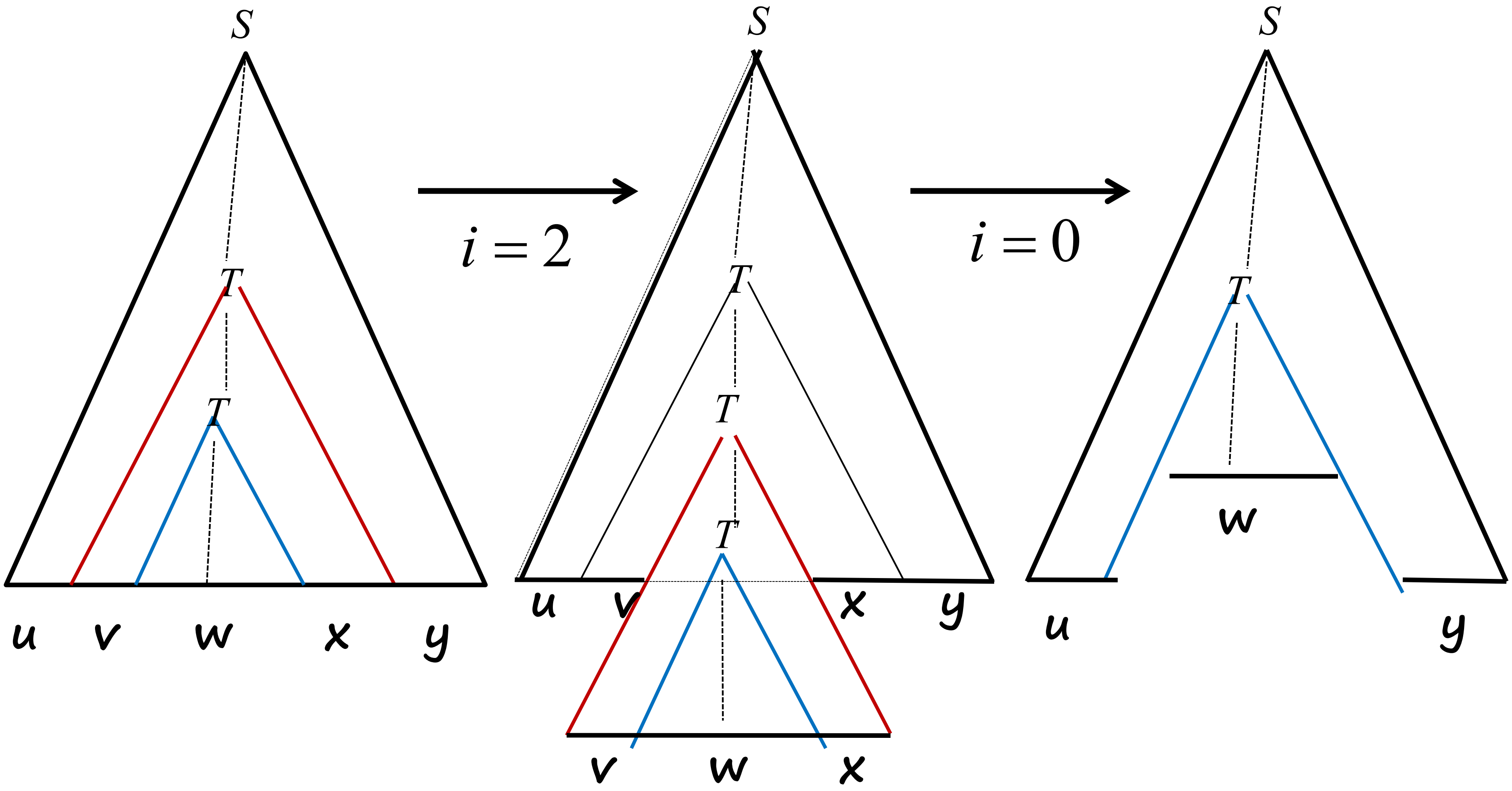
Pumping Lemma For CFL

- 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 $i \geq 0$, $uv^iwx^iy \in L$.
- This means that interesting context free languages (infinite ones) have a self-embedding property that is symmetric around some central area, unlike regular where the repetition has no symmetry and occurs at the start.

Pumping Lemma Proof

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

Visual Support of Proof



Lemma's Adversarial Process

- Assume L = {$a^n b^n c^n$ | 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| ≤ N, |vx|>0, and for all i≥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| ≤ 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∉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∉L
- CONTRADICTION, so L is NOT a CFL

Non-Closure

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

Max and Min of CFL

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

Complement of ww

- Let L = { ww | 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 → AB | BA ; A → XAX | a ; B → XBX | b
  X → a | 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∈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
- Σ is finite input alphabet
- Γ is finite set of stack symbols
- $\delta : Q \times \Sigma_e \times \Gamma_e \rightarrow 2^{Q \times \Gamma^*}$ is transition function
  - Note: Can limit stack push to Γ_e but it's equivalent!!
- $Z_0 \in \Gamma$ is an optional initial symbol on stack
- $F \subseteq Q$ is final set of states and can be omitted for some notions of a PDA

Notion of ID for PDA

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

Language Recognized by PDA

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

Top Down Parsing by PDA

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

- Give just one state, this is essentially stateless, except for stack

Top Down Parsing by PDA

E → E + T | T

T → T * F | F

F → (E) | Int

- δ(q,+,+) = {(q,λ)}, δ(q,*,*) = {(q,λ)},
- δ(q,Int,Int) = {(q,λ)},
- δ(q,(,() = {(q,λ)}, δ(q,),)) = {(q,λ)}
- δ(q,λ,E) = {(q,E+T), (q,T)}
- δ(q,λ,T) = {(q,T*F), (q,F)}
- δ(q,λ,F) = {(q,(E)), (q,Int)}

Bottom Up Parsing by PDA

- Given G = (V, Σ, R, S), define
  A = ({q,f}, Σ, Σ ∪ V ∪ {$}, δ, q, $, {f})
- $\delta(q,a,\lambda) = \{(q,a)\}$ for all $a \in \Sigma$, SHIFT
- $\delta(q,\lambda,\alpha^R) \supseteq \{(q,A)\}$ if $A \to \alpha \in R$, 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 → E + T | T

T → T * F | F

F → (E) | Int

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

Challenge

- Use the two recognizers on some sets of expressions like
  - 5 + 7 * 2
  - 5 * 7 + 2
  - (5 + 7) * 2

Converting a PDA to CFG

- Book has one approach; here is another
- Let A = (Q, Σ, Γ, δ, q_0, Z, F) accept L by empty stack and final state
- Define A' = ($Q \cup \{q_0', f\}$, Σ, $\Gamma \cup \{\$\}$, δ', q_0', \$, {f}) where
  - $\delta'(q_0', \lambda, \$) = \{(q_0, \text{PUSH}(Z))$ or in normal notation $\{(q_0, Z\$)\}$
  - δ' does what δ does but only uses PUSH and POP instructions, always reading top of stack
    Note1: we need to consider using the \$ for cases of the original machine looking at empty stack, when using λ for stack check. This guarantees we have top of stack until very end.
    Note2: If original adds stuff to stack, we do pop, followed by a bunch of pushes.
  - We add (f, λ) = (f, 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, Σ, R, S) where
  - V = {S} $\cup$ { <q, X, p> | q,p $\in$ Q, X $\in$ Γ }
  - R on next page

Rules for PDA to CFG

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

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 λ 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
      $\delta_0(q, a, X) \supseteq \{(q', \alpha)\}$ and
      $\delta_1(s,a) = s'$ (if $a=\lambda$ then $s' = s$).
  - Using the definition of derivations we see that
    $[<q_0,q_1>, w, \$] \vdash\!\!—^* [<t,s>, \lambda, \beta]$ in A_2 iff
    $[q_0, w, \$] \vdash\!\!—^* [t, \lambda, \beta]$ in A_0 and
    $[q_1, w] \vdash\!\!—^* [s, \lambda]$ in A_1
    But then $w \in \mathrm{F}(A_2)$ iff $t \in F_0$ and $s \in F_1$ iff $w \in \mathrm{F}(A_0)$ and $w \in \mathrm{F}(A_1)$

Substitution

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

More on Substitution

- Consider G'_f. If we limit derivations to the rules $R' = \{ A \rightarrow g(\alpha)$ where $A \rightarrow \alpha$ is in $R \}$ and consider only sentential forms over the $\cup_{a\in\Sigma} S_a$, then $S \Rightarrow^* S_{a1}\ S_{a2} \ldots S_{an}$ in G' iff $S \Rightarrow^*$ a1 a2 … an iff a1 a2 … an $\in \mathrm{L}(G)$. But, then $w \in \mathrm{L}(G)$ iff $f(w) \in \mathrm{L}(G_f)$ and, thus, $f(\mathrm{L}(G)) = \mathrm{L}(G_f)$.

- 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 <u>not</u> 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 $S \rightarrow \lambda$ so long as S never appears on the right hand side of any rule.

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

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

Phrase Structured Grammar

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

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

Assignment # 7

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) $L = \{ a^i b^j \mid j = i^2, i, j > 0 \}$
   b) $L = \{ a^i b^j c^k d^m \mid m + k = i + j \}$

2. Consider the context-free grammar **$G = \{ \{S\}, \{a,b\}, R, S \}$**
   R:
   $S \rightarrow a \mid b \mid a\ a \mid b\ b \mid a\ S\ a \mid b\ S\ b$
   Provide a proof that shows
   $L = \{ w \mid w \in \{a,b\}^+$ and w is a palindrome $\}$
   You will need to provide an inductive proof in both directions

Due: 10/26 (Thursday), 1:30PM (use Webcourses to turn in)

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

$Cb \rightarrow bC$ // Shuttle C over to a c

$Cc \rightarrow cc$ // 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 = { ww | w $\in \{0,1\}^+$ }

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

S → 00 | 11 | 0A<0> | aA<1> | 1A<1>

A → 0AZ | 1AX | 0Z | 1X

Z0 → 0Z	Z1 → 1Z	// Shuttle Z (for owe zero)
X0 → 0X	X1 → 1X	// Shuttle X (for owe one)
Z<0> → 0<0>	Z<1> → 1<0>	// New 0 must be on rhs of old 0/1's
X<0> → 0<1>	X<1> → 1<1>	// New 1 must be on rhs of old 0/1's
<0> → 0		// Guess we are done
<1> → 1		// Guess we are done

Midterm#2 Topics

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

Midterm#2 Topics

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

Midterm#2 Topics

- Context sensitive grammars and LBAs
  - Rules for CSG
  - Ability to shuttle symbols to preset places
  - Just basic definition of LBA
- Other
  - Guarantee closure based on substitution and intersection with Regular sets as seen in Exam#1 for Regular languages

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**. **Ø** denotes the empty set.
- (Decision) Problem -- A set of questions, each of which has answer “yes” or “no”.

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

Categorizing Problems (Sets) # 2

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

Immediate Implications

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

Slightly Harder Implications

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

- We will prove these later.

Existence of Undecidables

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

Hilbert's Tenth

Diophantine Equations are Unsolvable

One Variable Diophantine Equations are Solvable

Hilbert's 10th

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

Hilbert's 10^{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 $P(x_1,x_2,..,x_k) = 0$.

P(x) = 0 is Decidable

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

P(x) = 0 is Decidable

- Can bound the search to values of x in range [± n * (c_{max} / c_n)], where
  n = highest order exponent in polynomial
  c_{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 $P(x_1,x_2,..,x_k) = 0$.

Turing Machines

1st Model

A Linear Memory Machine

Textbook Description

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

Turing versus Post

- The Turing description just given requires you to write a new symbol and move off the current tape square
- 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.

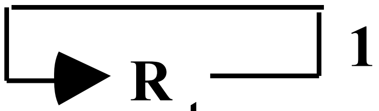
Base Machines

- 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

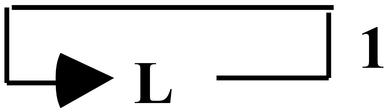
$\mathcal{R}$ -- move right to next 0 (not including current square)

…$\underline{?}$11…10… ⇒ …?11…1$\underline{0}$…



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

…011…1$\underline{?}$… ⇒ …$\underline{0}$11…1?…



Commentary on Machines

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

Computing with TMs

A reasonably standard definition of a Turing computation of some n-ary function F is to assume that the machine starts with a tape containing the n inputs, x1, … , xn in the form

$$\ldots 0 1^{x1} 0 1^{x2} 0 \ldots 0 1^{xn} \underline{0} \ldots$$

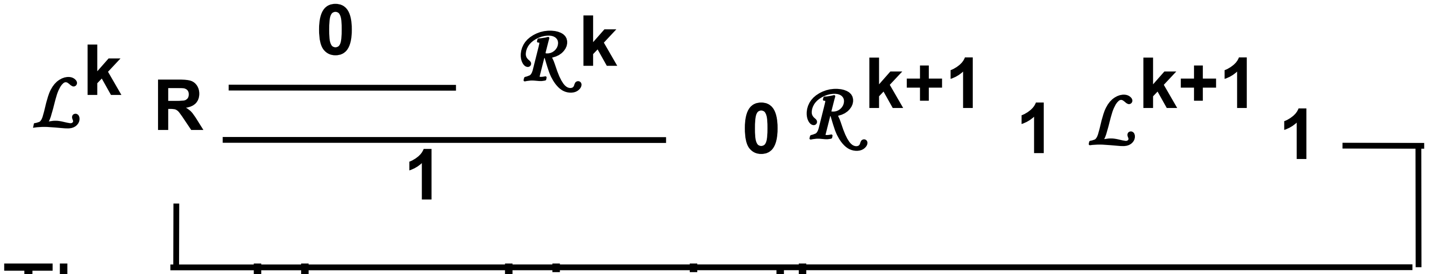
and ends with

$$\ldots 0 1^{x1} 0 1^{x2} 0 \ldots 0 1^{xn} 0 1^{y} \underline{0} \ldots$$

where y = F(x1, … , xn).

Addition by TM

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



$\mathcal{L}^k$ R 0 $\mathcal{R}^k$ 1 0 $\mathcal{R}^{k+1}$ 1 $\mathcal{L}^{k+1}$ 1

The add machine is then

$$C_2\ C_2\ \mathcal{L}\ 1\ \mathcal{R}\ L\ 0$$

Turing Machine Variations

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

Register Machines

2nd Model

Feels Like Assembly Language

Register Machine Concepts

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

Computing by Register Machines

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

Register Instructions

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

  $\mathbf{INC_r[i]}$ –

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

  $\mathbf{DEC_r[p, z]}$ –

  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 (r3 ← r1 + r2)

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

Limited Subtraction by RM

Subtraction (r3 ← r1 - r2, if r1≥r2; 0, otherwise)

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

Factor Replacement Systems

3rd Model

Deceptively Simple

Factor Replacement Concepts

- A factor replacement system (FRS) consists of a finite (ordered) sequence of fractions, and some starting natural number **x**.
- A fraction **a/b** is applicable to some natural number **x**, just in case **x** is divisible by **b**. We always chose the first applicable fraction (**a/b**), multiplying it times **x** to produce a new natural number **x*a/b**. The process is then applied to this new number.
- Termination occurs when no fraction is applicable.
- A factor replacement system partially computing **n**-ary function **F** typically starts with its argument encoded as powers of the first **n** odd primes. Thus, arguments **x1**,**x2**,…,**xn** are encoded as $\mathbf{3^{x1}5^{x2}\ldots p_n^{xn}}$. The result then appears as the power of the prime **2**.

Addition by FRS

Addition is $\mathbf{3^{x1}5^{x2}}$ becomes $\mathbf{2^{x1+x2}}$

or, in more details, $\mathbf{2^{0}3^{x1}5^{x2}}$ becomes $\mathbf{2^{x1+x2}\,3^{0}5^{0}}$

2 / 3

2 / 5

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

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

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

The previous rules would then be written

$\mathbf{3x} \rightarrow \mathbf{2x}$

$\mathbf{5x} \rightarrow \mathbf{2x}$

Limited Subtraction by FRS

Subtraction is $\mathbf{3^{x1}5^{x2}}$ becomes $\mathbf{2^{max(0,x1-x2)}}$

$$\mathbf{3 \cdot 5x \rightarrow x}$$

$$\mathbf{3x \rightarrow 2x}$$

$$\mathbf{5x \rightarrow x}$$

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

$\mathbf{3{\cdot}5x \rightarrow x}$

$\mathbf{3x \rightarrow 2x}$

$\mathbf{5x \rightarrow x}$

Starting with **135 =** $\mathbf{3^3 5^1}$, deterministically we get

$\mathbf{135 \Rightarrow 9 \Rightarrow 6 \Rightarrow 4 = 2^2}$

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

$\mathbf{135 \Rightarrow 9 \Rightarrow 6 \Rightarrow 4 = 2^2}$

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

$\mathbf{135 \Rightarrow 45 \Rightarrow 3 \Rightarrow 2 = 2^1}$

$\mathbf{135 \Rightarrow 45 \Rightarrow 15 \Rightarrow 1 = 2^0}$

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

$\mathbf{135 \Rightarrow 45 \Rightarrow 15 \Rightarrow 3 \Rightarrow 2 = 2^1}$

$\mathbf{135 \Rightarrow 45 \Rightarrow 9 \Rightarrow 6 \Rightarrow 4 = 2^2}$

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

…

This computes $\mathbf{2^z}$ where $\mathbf{0 \le z \le x_1}$. 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

$$\mathbf{2x} \rightarrow \mathbf{x}$$

$$\mathbf{2x} \rightarrow \mathbf{4x}$$

starting with the number **2**.

Sample RM and FRS

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

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

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

3*3 x → x

3 x → 2 x

Sample FRS

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

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

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

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

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

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

$11\ x \rightarrow 5\ x$

$5\ x \rightarrow x$

$7\ x \rightarrow x$

Primitive Recursive

An Incomplete Model

Basis of PRFs

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

$\mathbf{C_a(x_1,\ldots,x_n) = a}$	: constant functions
$\mathbf{I_i^n(x_1,\ldots,x_n) = x_i}$	: identity functions
	: aka projection
$\mathbf{S(x) = x+1}$	: an increment function

Building New Functions

- **Composition:**

  If **G**, **H_1**, … , **H_k** are already known to be primitive recursive, then so is **F**, where

  $$F(x_1,\dots,x_n) = G(H_1(x_1,\dots,x_n), \dots , H_k(x_1,\dots,x_n))$$

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

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

  $$F(0, x_1,\dots,x_n) = G(x_1,\dots,x_n)$$

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

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

Addition & Multiplication

Example: Addition

$+(0,y) = I_1^1(y)$

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

where $H(a,b,c) = S(I_3^3(a,b,c))$

Example: Multiplication

$*(0,y) = C_0(y)$

$*(x+1,y) = H(x,y,*(x,y))$

where $H(a,b,c) = +(I_2^3(a,b,c), I_3^3(a,b,c))$

$= b+c = y + *(x,y) = (x+1)*y$

Intuitive Composition

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

Intuitive Inductions

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

Basic Arithmetic

$x + 1$:

$x + 1 = S(x)$

$x – 1$:

$0 - 1 = 0$

$(x+1) - 1 = x$

$x + y$:

$x + 0 = x$

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

$x – y$: // limited subtraction

$x – 0 = x$

$x – (y+1) = (x–y) – 1$

2nd Grade Arithmetic

x * y:

$x * 0 = 0$

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

x!:

$0! = 1$

$(x+1)! = (x+1) * x!$

Basic Relations

$x == 0$:

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

$x == y$:

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

$x \le y$:

- $x{\le}y = (x - y) == 0$

$x \ge y$:

- $x{\ge}y = y{\le}x$

$x > y$:

- $x{>}y = \sim(x{\le}y)$ /* See ~ on next page */

$x < y$:

- $x{<}y = \sim(x{\ge}y)$

Basic Boolean Operations

~x:

~x = x==0

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

~(x==0)

x && y:

x&&y = signum(x*y)

x || y:

x||y = ~((x==0) && (y==0))

Definition by Cases

One case

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

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

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

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

Bounded Minimization 1

$\mathbf{f(x) = \mu\, z\ (z \le x)\ [\ P(z)\ ]}$ if ∃ such a $\mathbf{z}$,

$\mathbf{= x+1}$, otherwise

where $\mathbf{P(z)}$ is primitive recursive.

Can show $\mathbf{f}$ is primitive recursive by

$$
\begin{array}{lcll}
\mathbf{f(0)} & \mathbf{=} & \mathbf{1\text{-}P(0)} & \\
\mathbf{f(x+1)} & \mathbf{=} & \mathbf{f(x)} & \text{if } \mathbf{f(x) \le x} \\
 & \mathbf{=} & \mathbf{x+2\text{-}P(x+1)} & \text{otherwise}
\end{array}
$$

Bounded Minimization 2

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

$= x$, otherwise

where **P(z)** is primitive recursive.

Can show **f** is primitive recursive by

$f(0) = 0$

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

Intermediate Arithmetic

x // y:

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

x//(y+1) = μ z (z<x) [(z+1)*(y+1) > x]

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

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

Primality

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

firstfactor(x) = $\mu\, z\ (2 \le z \le x)\ [\, z|x\,]$,
0 if none

isPrime(x):

isPrime(x) = firstFactor(x) == x && (x>1)

prime(i) = i-th prime:

prime(0) = 2

prime(x+1) = $\mu\, z(\mathrm{prime}(x) < z \le \mathrm{prime}(x)!+1)[\mathrm{isPrime}(z)]$

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

Exponents

x^y:

$x^0 = 1$

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

exp(x,i): the exponent of **p_i** in number **x.**

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

Pairing Functions

- **pair(x,y) = <x,y> = 2^x (2y + 1) – 1**

- with inverses

  $<z>_1$ = exp(z+1,0)

  $<z>_2$ = (((z + 1) // 2 $^{<z>_1}$) – 1) // 2

- These are very useful and can be extended to encode **n**-tuples

  <x,y,z> = <x, <y,z> > (note: stack analogy)

Pairing Function is 1-1 Onto

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

Union/Find

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

The μ Operator

- Minimization:

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

  $$\mathbf{F(x1,\ldots,xn) = \mu y\ (G(y,x1,\ldots,xn) == 1)}$$

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

Universal Machine

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

Encoding FRS

- Let **(n, ((a$_1$,b$_1$), (a$_2$,b$_2$), … ,(a$_n$,b$_n$))** be some factor replacement system, where **(a$_i$,b$_i$)** means that the **i**-th rule is

  a$_i$x → b$_i$x

- Encode this machine by the number **F**,

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

Simulation by Recursive # 1

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

  RULE(F, x) = μ z (1 ≤ z ≤ exp(F, 0)+1) [exp(F, 2*z-1) | x]
- Note: if **x** is divisible by **a_i**, and **i** is the least integer for which this is true, then **exp(F,2*i-1) = a_i** where **a_i** is the number of prime factors of **F** involving **p_{2i-1}**. Thus, **RULE(F,x) = i**.

  If x is not divisible by any **a_i**, **1≤i≤n**, then **x** is divisible by **1**, and **RULE(F,x)** returns **n+1**. That's why we added **p_{2n+1} p_{2n+2}**.
- Given the function **RULE(F,x)**, we can determine **NEXT(F,x)**, the number that follows **x**, when using **F**, by

  NEXT(F, x) = (x // exp(F, 2*RULE(F, x)-1)) * exp(F, 2*RULE(F, x))

Simulation by Recursive # 2

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

CONFIG(F, x, 0) = x

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

- The number of the configuration on which **F** halts is

HALT(F, x) = μ y [CONFIG(F, x, y) == CONFIG(F, x, y+1)]

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

Simulation by Recursive # 3

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

Undecidability

We Can’t Do It All

Cantor and Infinities

The previous "brash" statement (page 24) suggests there are at least two infinite cardinals, $|\boldsymbol{N}|$ and $|\boldsymbol{R}|$. Furthermore, $|\boldsymbol{N}|$ is a countable cardinal and $|\boldsymbol{R}|$ is an uncountable cardinal. In fact there are infinitely many distinct cardinal numbers representing infinite sets!

In addition to these facts, Cantor proved that there is a smallest infinite cardinal number. He designated this smallest infinite cardinal number, $\aleph_0$, named "aleph-null"; aleph is a symbol in the Hebrew alphabet. He further showed that given any cardinal number, $\aleph_k$, there is a next smallest cardinal number, $\aleph_{k+1}$.

Cantor was able to prove that $|\ \boldsymbol{N}\ | = \aleph_0$, and although many mathematicians believe that $\aleph_1 = |\boldsymbol{R}|$, this has never been proven from the axioms of mathematical set theory.

How Many Infinities?

- The theorem stated and proven next is due to Cantor and gives us a mechanism for defining two sets of distinctly different cardinality (one being strictly larger than the other). By inductively applying Cantor's theorem it follows that there are infinitely many cardinal numbers denoting the sizes of infinite sets. Cantor's theorem uses the power set of a given set.

Cantor’ s Theorem

Theorem (Cantor). Let S be any set. Then $|S| < |\mathcal{P}(S)|$.

Proof.

Case1: Suppose $S = \emptyset$. Then $\mathcal{P}(S) = \{\emptyset\}$. Since $|S| = 0$ and $|\mathcal{P}(S)| = 1$, the result holds.

Case2: Assume $S \neq \emptyset$.

(a) First we show that $|S| \leq |\mathcal{P}(S)|$.

To show this we must find an injection, f, from S to $\mathcal{P}(S)$.

Consider $f(x) = \{x\}$. Clearly, $f(x) \in \mathcal{P}(S)$ for all $x \in S$.

Furthermore, if $x \neq y$, then $f(x) = \{x\} \neq \{y\} = f(y)$.

Thus f is the desired function and we may conclude that $|S| \leq |\mathcal{P}(S)|$.

(b) Next we wish to show $|S| \neq |\mathcal{P}(S)|$. We do this by contradiction.

Assume $|S| = |\mathcal{P}(S)|$, then by definition of equality of cardinal numbers, there is a function, f, that is 1-1 and onto from S to $\mathcal{P}(S)$.

Define $Z = \{ x \in S \mid x \notin f(x) \}$. Clearly, Z is a subset (possibly empty) of S.

Therefore there is a $y \in S$ such that $f(y) = Z$. This follows from our assumption that f is onto $\mathcal{P}(S)$. Then either $y \in Z$ or $y \notin Z$.

(b.1) Suppose $y \in Z$, then by definition of Z, $y \notin f(y) = Z$; a contradiction.

(b.2) Suppose $y \notin Z$, then by definition of Z, $y \in f(y) = Z$; a contradiction.

Since the existence of f led to this logical absurdity, we must conclude that f cannot exist and thus $|S| = |\mathcal{P}(S)|$ is false. This establishes (b).

(a) and (b) together imply $|S| < |\mathcal{P}(S)|$.

Corollaries

- If **$|S| = |\mathbb{N}|$**, then **$|\mathcal{P}(S)| > |\mathbb{N}| = \aleph_0$**.

- There are sets whose cardinalities are greater than $\aleph_0$. These sets are uncountably infinite, whereas those that correspond to $\mathbb{N}$ are countably infinite.
- Note that a set can be countable and yet there is no effective way to describe its correspondence with $\mathbb{N}$. Look back and you will see that the definition just says that an injective function exists, not that this function is actually computable.

Cardinalities of Z and Q

1. We show that **|** N **| = |** Z **|**.
   | N **| ≤ |** Z **|**: Define **g:** $N \rightarrow Z$ as follows: **g(i) = i**
   | Z **| ≤ |** N **|**: Define **f:** $Z \rightarrow N$ as follows:

$$f(x) = \begin{cases} 0 & \text{, if x} = 0 \\ 2\text{x} - 1, & \text{if x} > 0 \\ -2\text{x} & \text{, if x} < 0 \end{cases}$$

x=	0	1	-1	2	-2
f(x)=	0	1	2	3	4

2. To show **|** N **| = |** Q **|** we develop the proof in two steps:
   (a) Lemma – prove that **|A|** ≤ **|S|** for every subset **A** of **S**.
   Note: This is what we did for **|** N **| ≤ |** Z |

   (b) Prove that **|** $N \times N$ **| = |** N **|**.

|Subset| ≤ |Parent Set|

Lemma A. **|A|** ≤ **|S|**, for every subset **A** of **S**.

Proof. Let **A** be a subset of **S**. To establish that **|A|** ≤ **|S|** we need to find a 1-1 function from **A** into **S**. The identity function, **f(x) = x**, is the desired function; clearly, if **x** ≠ **y**, then **f(x) = x** ≠ **y = f(y)**. Since, **f(x)** ∈ **S**, for every **x** in **A**, the lemma is proved.

$| N \times N | = | N |$

Lemma B. $| N \times N | = | N |$.

Proof. Let **S = $N \times N$ = {(k,j) | k,j ∈ N}**. Define the function, f((k,j)) = ((k+j)(k+j+1))/2 + j. Clearly f is a function, since the defining expression is single-valued.

Furthermore, ∀ k,j ∈ N, f((k,j)) ≥ 0. We have to show that f is 1-1 and onto N.
To show f is 1-1, let (k, j) and (k', j') be two distinct elements of S.
There are two cases to consider. (a) k+j = k'+j', or (b) k+j < k'+j' (or k'+j' < k+j).

Assume (a). Then f((k,j)) – f((k',j')) = j – j' (we can assume without loss of generality that j-j' ≥ 0). If j-j' = 0, then j = j'. Thus k+j = k'+j' implies k = k', but this contradicts our assumption that (k,j) and (k',j') are distinct elements of S. Thus we must assume that j-j' > 0. It follows immediately that f((k,j)) ≠ f((k',j').

Assume (b). Then we can assume k+j < k'+j' = k+j+a, for some a > 0. Now suppose f(k',j')) = f((k,j)). Substituting k+j+a for k'+j' in the formula for f((k',j')) and equating to f((k,j)), and doing the algebra we arrive at j = aj + y, where y is some positive number. Clearly this relation cannot hold for any non-negative j and a > 0. We must conclude that f((k,j)) ≠ f((k',j'). Thus f is 1-1.

To show that f is onto N, we need to show that given any m ≥ 0, there is a (k,j) such that f((k,j)) = m. Let x be the largest non-negative integer such that x(x+1)/2 ≤ m. It follows that (x+1)(x+2)/2 > m. Now choose j = m - x(x+1)/2 and k = x-j. It follows that f((k,j)) = m.

Proof That $|\boldsymbol{N}| = |\boldsymbol{Q}|$

By definition, $\boldsymbol{Q} = \{ (a,b) \mid a \in \boldsymbol{Z} \text{ and } b \in \boldsymbol{Z}^+ \}$

$|\boldsymbol{Q}| \leq |\boldsymbol{N}|$.

$\boldsymbol{Q} \subseteq \boldsymbol{Z} \times \boldsymbol{N}$. Thus $|\boldsymbol{Q}| \leq |\boldsymbol{Z} \times \boldsymbol{N}|$ by Lemma A.

But $|\boldsymbol{Z} \times \boldsymbol{N}| = |\boldsymbol{N} \times \boldsymbol{N}|$ using an argument similar to that showing $|\boldsymbol{Z}| = |\boldsymbol{N}|$. (Define g by g(a,b) = (*f*(a),b)) where *f* is the function used to map $\boldsymbol{Z}$ to $\boldsymbol{N}$.)

By Lemma B it follows that $|\boldsymbol{Q}| \leq |\boldsymbol{N}|$.

$|\boldsymbol{N}| \leq |\boldsymbol{Q}|$.

Define f(a) = (a,1). This is a 1-1 mapping from $\boldsymbol{N}$ into $\boldsymbol{Q}$, showing $|\boldsymbol{Q}| \leq |\boldsymbol{N}|$.

Thus, $|\boldsymbol{N}| = |\boldsymbol{Q}|$.

Computable Languages 1

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

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

Computable Languages 2

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

Programming Languages

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

Classic Unsolvable Problem

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

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

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

Some terminology

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

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

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

Finally, we define the domain of f as $\{x \mid f(x)\downarrow\}$.
The range of f is $\{y \mid$ there exists an x, $f(x)\downarrow$ and $f(x) = y\}$.

Numbering Procedures

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

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

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

The universal machine

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

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

Halting Problem (A_{TM})

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

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

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

pair(x,y) = <x,y> = 2^x (2y + 1) – 1

with inverses

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

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

The Contradiction

Now if Halt exist, then so does Disagree, where

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

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

Disagree(d) is defined ⇔ Halt(d,d) = 0

⇔ $\varphi_{\mathbf{d}}(\mathbf{d})$ is undefined

⇔ Disagree(d) is undefined

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

Halting (A_{TM}) is recognizable

While the Halting Problem is not solvable, it is 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.

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

Enumeration Theorem

- Define

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

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

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

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

Non-re Problems

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

  $F(x) = F_x$

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

The Contradiction

- Define $G(x) = Univ (F(x) , x) + 1 = \varphi_{F(x)}(x) = F_x(x) + 1$
- But then G is itself an algorithm. Assume it is the g-th one

  $F(g) = F_g = G$

  Then, $G(g) = F_g(g) + 1 = G(g) + 1$
- But then G contradicts its own existence since G would need to be an algorithm.
- This cannot be used to show that the effective procedures are non-enumerable, since the above is not a contradiction when G(g) is undefined. In fact, we already have shown how to enumerate the (partial) recursive functions.

The Set TOTAL

- The listing of all algorithms can be viewed as
  TOTAL = $\{ f \in \boldsymbol{N} \mid \forall x\ \varphi_f (x)\downarrow \}$
- We can also note that
  TOTAL = $\{ f \in \boldsymbol{N} \mid W_f = \boldsymbol{N} \}$, where W_f is the domain of φ_f
- Theorem: TOTAL is not re.
  Proof: Shown earlier.

Consequences

- To capture all the algorithms, any model of computation must include some procedures that are not algorithms.

- Since the potential for non-termination is required, every complete model must have some form of iteration that is potentially unbounded.

- This means that simple, well-behaved for-loops (the kind where you can predict the number of iterations on entry to the loop) are not sufficient. While type loops are needed, even if implicit rather than explicit.

Insights

Non-re nature of algorithms

- No generative system (e.g., grammar) can produce descriptions of all and only algorithms
- No parsing system (even one that rejects by divergence) can accept all and only algorithms

- Of course, if you buy Church’s Theorem, the set of all procedures can be generated. In fact, we can build an algorithmic acceptor of such programs.

Many unbounded ways

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

Non-determinism

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

Reducibility

Reduction Concepts

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

Reduction Example#1

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

Reduction Examples #2 & #3

In all cases below we are assuming our variables are over $\aleph$.

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

- Show ZERO = $\{ f \mid \forall x\ \varphi_f\,(x) = 0 \}$ is unsolvable.
  $<f,x> \in$ HALT iff $g(y) = \varphi_f\,(x) - \varphi_f\,(x)$ is zero for all y.
  Thus, $<f,x> \in$ HALT iff $g \in$ ZERO (really the index of g).
  A solution to ZERO implies one for HALT, so ZERO is unsolvable.

- Show ZERO = $\{ f \mid \forall x\ \varphi_f\,(x) = 0 \}$ is non-re.
  $<f> \in$ TOTAL iff $h(x) = \varphi_f\,(x) - \varphi_f\,(x)$ is zero for all x.
  Thus, $f \in$ TOTAL iff $h \in$ ZERO (really the index of h).
  A semi-decision procedure for ZERO implies one for TOTAL, so ZERO is non-re.

Reduction and Equivalence

m-1, 1-1, Turing Degrees

Many-One Reduction

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

Many-One Degrees

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

One-One Reduction

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

One-One Degrees

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

Turing (Oracle) Reduction

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

Turing Degrees

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

Complete re Sets

- 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 HALT is an re complete set (in regard to 1-1, m-1 and Turing reducibility).
- The re complete degree (in each sense of degree) sits at the top of the lattice of re degrees.

The Set Halt = K_0

- Halt = K_0 = { <f, x> | φ_f (x) is defined }
- Let A be an arbitrary re set. By definition, there exists an effective procedure φ_a, such that dom(φ_a) = A. Put equivalently, there exists an index, a, such that A = W_a.
- $x \in A$ iff $x \in \mathrm{dom}(\varphi_a)$ 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, Halt = K_0, is an re (1-1, m-1, Turing) complete set.

The Set K

- K = { f | $\varphi_f(f)$ is defined }
- Define $f_x(y) = \varphi_f(x)$. That is, $\forall y\ f_x(y) = \varphi_f(x)$. Let the index of f_x be f_x. (Yeah, that's overloading.)

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

- The above provides a 1-1 function that reduces K_0 to K.
- Since K_0 is an re (1-1, m-1, Turing) complete set and K is re, then K is also re (1-1, m-1, Turing) complete.

Reduction and Rice’s

Either Trivial or Undecidable

- Let P be some set of re languages, e.g. P = { L | L 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 | dom [x] is in P (has property P) }

is undecidable.

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

Rice's Proof-1

Proof: We will assume, *wlog*, that **P** does not contain **Ø**. 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 $\mathbf{K_0}$.

Rice's Proof-2

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

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

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

Rice's Proof-3

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

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

I/O Properties

- An I/O property, **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 $\varphi_{\mathbf{f}}$ and $\varphi_{\mathbf{g}}$ converge on the same inputs and, when they converge, produce precisely the same result, then both **f** and **g** must have property **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 $\mathbf{P}$ be some non-trivial I/O property of the indices of recursive functions. Then

$$\mathbf{S_P = \{\ x \mid \varphi_x \text{ has property } P)\ \}}$$

is undecidable.

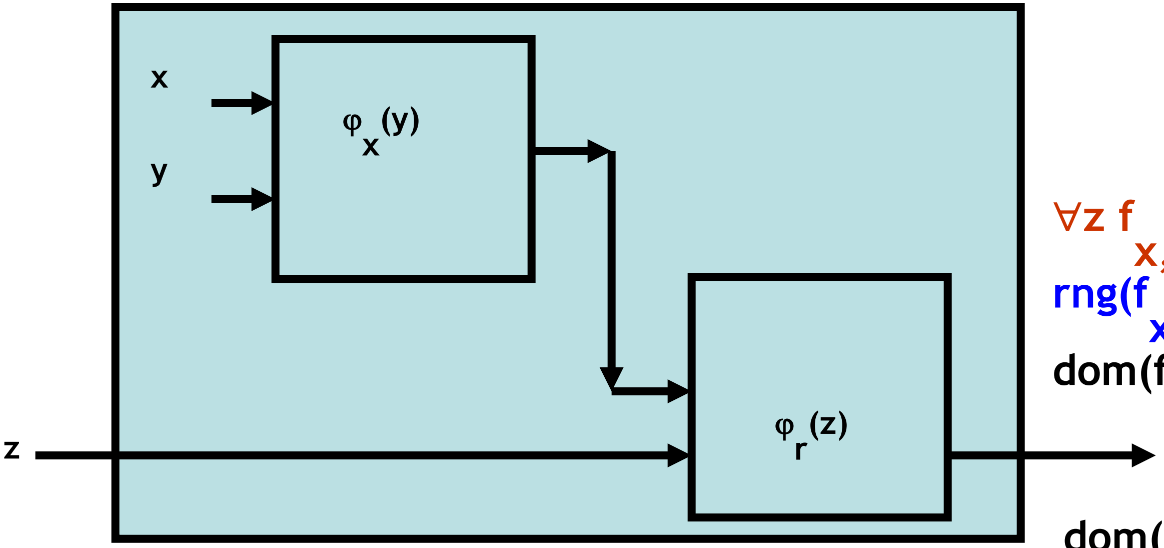
Note that membership in $\mathbf{S_P}$ is based purely on the input/output behavior of a function, not on any aspect of its implementation.

Strong Rice's Proof

- Given **x**, **y**, **r**, where **r** is in the set
  $\mathbf{S_{P}}$.= {**f** | $\boldsymbol{\varphi_f}$ has property P},
  define the function
  $\mathbf{f_{x,y,r}(z) = \varphi_x(y) - \varphi_x(y) + \varphi_r(z)}$.

- $\mathbf{f_{x,y,r}(z) = \varphi_r(z)}$ **if** $\boldsymbol{\varphi_x(y)\downarrow}$ **;** $\mathbf{= \phi}$ **if** $\boldsymbol{\varphi_x(y)\uparrow}$.
  Thus, $\boldsymbol{\varphi_x(y)\downarrow}$ iff $\mathbf{f_{x,y,r}}$ has property P, and so
  $\mathbf{K_0 \le S_{P}}$.

Rice’s Picture Proof



$\forall z\ f_{x,y,r}(z)=\varphi_r(z)$ If $\varphi_x(y)\downarrow$

$rng(f_{x,y,r})=rng(\varphi_r)$ If $\varphi_x(y)\downarrow$

$dom(f_{x,y,r})=dom(\varphi_r)$ If $\varphi_x(y)\downarrow$

$dom(f_{x,y,r})=\phi$ If $\varphi_x(y)\uparrow$

$rng(f_{x,y,r})=\phi$ If $\varphi_x(y)\uparrow$

$\exists z\ f_{x,y,r}(z)\neq\varphi_r(z)$ If $\varphi_x(y)\uparrow$

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 re sets are undecidable

a) $L = \emptyset$

b) L is finite

c) L is a regular set

d) L is a context-free set

Assignment # 8

Known Results:

HALT = { <f,x> | f(x)↓ } is re (semi-decidable) but undecidable

TOTAL = { f | ∀x f(x)↓ } is non-re (not even semi-decidable)

1. Use reduction from **HALT** to show that one cannot decide **HasExp**, where
   HasExp = { f | for some x, y, x<y, f(y) = 2^f(x) }
2. Show that **HasExp** reduces to **HALT**. (1 plus 2 show they are equally hard)
3. Use Reduction from **TOTAL** to show that **IsExp** is not even re, where
   IsExp = { f | for all x, there is some y, x<y, f(y) > 2^f(x) }
4. Show **IsExp** reduces to **TOTAL**. (3 plus 4 show they are equally hard)
5. Use Rice's Theorem to show that **HasExp** is undecidable
6. Use Rice's Theorem to show that **IsExp** is undecidable

Due: Thursday, 11/20, 12:00PM (use Webcourses to turn in)

Recursively Enumerable

Properties of re Sets

Definition of re

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

  $\mathbf{S} \subseteq \mathcal{N}$ is semi-decidable iff there exists a partially computable function **g** where

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

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

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

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

STP Predicate

- **STP(f, x1,…,xn, t)** is a predicate defined to be true iff φ_f**(x1,…,xn)** converges in at most **t** steps.
- **STP** can be shown to be a simple algorithm. Consider, for instance, a universal machine (interpreter) that is told the maximum number of step to simulate.

Semi-Decidable Implies re

Theorem: Let **S** be semi-decided by **G_S**. Assume **G_S** is the **g_S** function in our enumeration of effective procedures. If **S = Ø** then **S** is re by definition, so we will assume wlog that there is some ***a*** ∈ **S**. Define the enumerating algorithm **F_S** by

$$F_S(<x,t>) = \quad x * STP(g_s, x, t) + a * (1-STP(g_s, x, t))$$

Note: **F_S** is primitive recursive and it enumerates every value in **S** infinitely often.

re Implies Semi-Decidable

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

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

This achieves our result as the domain of ψ_S is the range of $\mathbf{F_S}$, or empty if **S == Ø**.

Domain of a Procedure

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

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

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

Recursive Implies re

Theorem: Recursive implies re.

Proof: **S** is recursive implies there is an algorithm $\mathbf{f_S}$ such that

$$\mathbf{S = \{ x \in \mathit{N} \mid f_s(x) == 1 \}}$$

Define $\mathbf{g_s(x) = \exists y\ (f_s(x) == 1)}$

Clearly

$$\begin{aligned}\mathbf{dom(g_s)} &\mathbf{= \{x \in \mathit{N} \mid g_s(x)\downarrow\}} \\ &\mathbf{= \{ x \in \mathit{N} \mid f_s(x) == 1 \}} \\ &\mathbf{= S}\end{aligned}$$

Related Results

Theorem: **S** is re iff **S** is semi-decidable.

Proof: That’s what we proved.

Theorem: **S** and **~S** are both re (semi-decidable) iff **S** (equivalently **~S**) is recursive (decidable).

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

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

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

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

re Characterizations

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

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

Quantification#1

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

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

Quantification#2

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

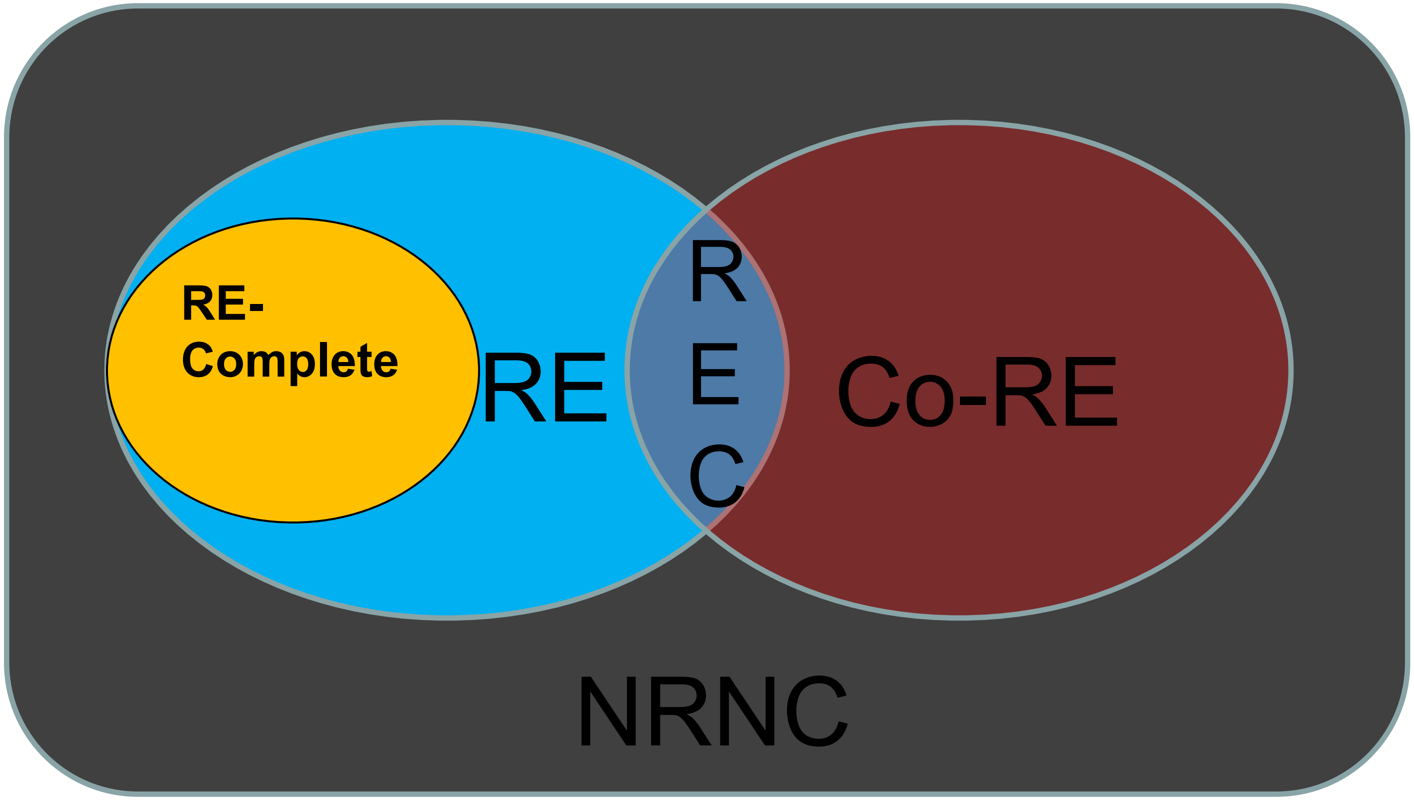
Quantification#3

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

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

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

UNIVERSE OF SETS



NR (non-recursive)

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

Sample Question#1

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

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

   Note: **STP(f, x, s)** is true iff **φ_f(x)** converges in **s** or fewer steps and, if so, **VALUE(f, x, s) = φ_f(x)**.

Sample Questions#2,3

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

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

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

Sample Question#4

4. Let **P = { f | ∃ x [STP(f, x, x)] }**. Why does Rice's theorem not tell us anything about the undecidability of **P**?

Sample Question#5

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

   E = { z | z = x + y, where x ∈ S and y ∈ T }.

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

   (a) Can **E** be non re?

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

   (c) Can **E** be recursive?

Assignment # 9

1. Use quantification of an algorithmic predicate to estimate the complexity (decidable, re, co-re, non-re) of each of the following, (a)-(d):
   a) **{ f | f is a Fibonacci function, i.e. f(0)=f(1)=1 and f(x+2)=f(x)+f(x+1) }**
   b) **(f | |range(f)| > 1 }**.
   c) **{ <f,x> | if f(x) converges, it does so in more than (2^x) units of time }**
   d) **{ f | f(x) = f(x+1) for at least one value of x }**
2. Let set **A** be recursive, and let **B** be re non-recursive.
   Consider **C = B – A** . For (a)-(c), either show sets **A** and **B** with the specified property or demonstrate that this property cannot hold.
   a) **Can C be recursive?**
   b) **Can C be re non-recursive (undecidable)?**
   c) **Can C be non-re?**

Due: Thursday, 11/30, 1:30PM (use Webcourses to turn in)

Rewriting Systems

Semi-Thue Systems

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

Simulating Turing Machines

- Basically, we need at least one rule for each 4-tuple in the Turing machine's description.
- The rules lead from one instantaneous description to another.
- The Turing ID $\alpha q a \beta$ is represented by the string $h\alpha q a \beta h$, a being the scanned symbol.
- The tuple q a b s leads to
  qa → 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 $h1^x q_1 0h$, seeking to accept x (enter q_0) or reject (run forever).
- Add rules
  - $q_0 a \rightarrow q_0$ $\forall a \in \{0,1\}$
  - $bq_0 \rightarrow q_0$ $\forall b \in \{0,1\}$

- 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:
- $h1^x q_1 0h \Rightarrow_{\Sigma(M)}^* hq_0h$ iff $x \in \mathcal{L}(M)$.
- $hq_0h \Rightarrow_{\Pi(M)}^* h1^x q_1 0h$ iff $x \in \mathcal{L}(M)$.
- $hq_0h \Leftrightarrow_{\Sigma\,(M)}^* h1^x q_1 0h$ iff $x \in \mathcal{L}(M)$.
  - This is called a Thue system where rules can be applied in either direction ($\alpha \leftrightarrow \beta$)
- Can recast both Semi-Thue and Thue Systems to ones over alphabet {a,b} or {0,1}. That is, a binary alphabet is sufficient for undecidability.

More on Grammars

Grammars and re Sets

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

Post Correspondence Problem

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

ST(Word) ≤ PCP

- Start with Semi-Thue System
  - aba → ab; a → aa; b → a
  - Instance of word problem: bbbb ⇒*? aa
- Convert to PCP

– [bbbb*	ab	$\underline{\text{ab}}$	aa	$\underline{\text{aa}}$	a	$\underline{\text{a}}$	]
[	$\underline{\text{aba}}$	aba	$\underline{\text{a}}$	a	$\underline{\text{b}}$	b	$\underline{*}$aa]
– And	*	$\underline{*}$	a	$\underline{\text{a}}$	b	$\underline{\text{b}}$	
	$\underline{*}$	*	$\underline{\text{a}}$	a	$\underline{\text{b}}$	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 $\underline{*}W_f]$
- Bottom row matches W_i, while top matches W_{i+1} (one is underscored)

PCP is undecidable

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

Ambiguity of CFG

- Arbitrary instance of PCP,
  $P = (\Sigma, n, ((\, x_1, \ldots, x_n\,), (\, y_1, \ldots, y_n\,)\,)$
- **$G = (\{S,A,B\}, \Sigma, R, S)$, where R is:**

  $S \rightarrow A \mid B$

  $A \rightarrow x_i\, A\, [i] \mid x_i\, [i]$ $\quad$ **$1 \le i \le n$**

  $B \rightarrow y_i\, B\, [i] \mid y_i\, [i]$ $\quad$ **$1 \le i \le n$**

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

  $B \Rightarrow^* y_{i_1} \ldots y_{i_k}\, [i_k] \ldots [i_1]$ $\quad$ **$k > 0$**
- Ambiguous if and only if there is a solution to this PCP instance, **P**.

Intersection of CFLs

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

Non-emptiness of CSL

- Arbitrary instance of PCP,
  $P = (\Sigma, n, ((x_1, \ldots, x_n), (y_1, \ldots, y_n)))$
- $G = (\{S,T\} \cup \Sigma, \{*\}, R, S)$, where R is:

$$\begin{array}{ll}
S & \rightarrow x_i\ S\ y_i^R \mid x_i\ T\ y_i^R \quad 1 \le i \le n \\
a\ T\ a & \rightarrow *\ T\ * \\
*\ a & \rightarrow a\ * \\
a\ * & \rightarrow *\ a \\
T & \rightarrow *
\end{array}$$

CSG Produces Something

- Our only terminal in previous grammar is $*$. We get strings of form $*^{2j+1}$, for some **j**'s if and only if there is a solution to this PCP instance. Get **Ø** otherwise.

- Thus, **P** has a solution iff
  - **L(G) ≠ Ø**
  - **L(G) is infinite**

Traces and Grammars

Traces (Valid Computations)

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

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

  where $X_i \Rightarrow X_{i+1}$ $0 \le i < k$, X_0 is a starting configuration and X_k is a terminating configuration.
- We allow some laxness, where the configurations might be encoded in a convenient manner. Many texts show that a context free grammar can be devised which approximates traces by either getting the even-odd pairs right, or the odd-even pairs right. The goal is then to intersect the two languages, so the result is a trace. This then allows us to create CFLs **L1** and **L2**, where $L1 \cap L2 \neq \emptyset$, just in case the machine has an element in its domain. Since this is undecidable, the non-emptiness of the intersection problem is also undecidable. This is an alternate proof to one we already showed based on PCP.

Quotients of CFLs (concept)

Let L1 = L(G1) = { $ # Y_0 # Y_1 # Y_2 # Y_3 # … # Y_{2j} # Y_{2j+1} # }
where $Y_{2i} \Rightarrow Y_{2i+1}$, $0 \le i \le j$.
This checks the even/odd steps of an even length computation.

Now, let L2 = L(G2) = {X_0 $ # X_0 # X_1 # X_2 # X_3 # X_4 # … # X_{2k-1} # X_{2k}# Z_0 #}
where $X_{2i-1} \Rightarrow X_{2i}$, $1 \le i \le k$ 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 L2 / L1 . The only ways a member of L1 can match a final substring in L2 is to line up the $ signs. But then they serve to check out the validity and termination of the computation. Moreover, the quotient leaves only the starting point (the one on which the machine halts.) Thus,

L2 / L1 = { X_0 | the system halts}.

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

Finish Quotient

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

L2 / L1 = { X | 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 (PSG)

- Here, it is actually easier to show a simulation of a Turing machine than of a Factor System.
- Assume we are given some machine M, with Turing table T (using Post notation). We assume a tape alphabet of Σ that includes a blank symbol B.
- Consider a starting configuration C0. Our rules will be

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

CSG and Undecidability

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

S'	$\rightarrow$	**e S e**	
D x	$\rightarrow$	**x D**	**when x $\in V \cup T$**
D e	$\rightarrow$	**e d**	**push the delete characters to far right**
α	$\rightarrow$	β	**where $\alpha \rightarrow \beta \in P$ and $\|\alpha\| \leq \|\beta\|$**
α	$\rightarrow$	βD^k	**where $\alpha \rightarrow \beta \in P$ and $\|\alpha\| - \|\beta\| = k > 0$**

- Clearly, L(G') = { e w e d^m | w $\in$ L(G) and m≥0 is some integer }
- For each w $\in$ L(G), we cannot, in general, determine for which values of m, e w e dm $\in$ L(G'). We would need to ask a potentially infinite number of questions of the form
"does e w e d^m $\in$ L(G')" to determine if w $\in$ L(G). That's a semi-decision procedure.

Some Consequences

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

Undecidability

- Is L $=\varnothing$, 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 $L_1\cap L_2=\varnothing$ 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 ~L CFL, for CFL, L?

More Undecidability

- Is CFL, L, ambiguous?
- Is L=L^2, L a CFL?
- Is L_1/L_2 finite, L_1 and L_2 CFLs?
- Membership in L_1/L_2, L_1 and L_2 CFLs?

ST(Word) ≤ PSL(Membership)

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

  G: $S \rightarrow h1^xq_10h$

  $hq_0h \rightarrow T$

  $T \rightarrow aT$

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

Consequences for PSG

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

L = Σ*?

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

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

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

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

- Question: Does $L \bullet L$ get us anything new?
  - i.e., Is $L \bullet L = L$?
- Membership in a 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 $\Sigma^* \subseteq L^* = \cup_{n\geq 0} L^n \subseteq L$
  - first inclusion follows from (1); second from (2)

Computational Complexity

Limited to Concepts of P and NP

COT6410 covers much more

Research Territory

Decidable – vs – Undecidable
(area of Computability Theory)

Exponential – vs – polynomial
(area of Computational Complexity)

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

Decision vs Optimization

Two types of problems are of particular interest:

Decision Problems ("Yes/No" answers)

Optimization problems ("best" answers)

(there are other types)

Natural Pairs of Problems

Interestingly, these usually come in pairs

a *decision* problem, and

an *optimization* problem.

Equally easy, or equally difficult, to solve.

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

Very Hard Problems

Some problems have no algorithm (e. g., Halting Problem.)

No mechanical/logical procedure will ever solve all instances of any such problem!!

Some problems have only exponential algorithms (provably so – they must take at least order $\mathbf{2^n}$ steps) So far, only a few have been proven, but there may be many. We suspect so.

Easy Problems

Many problems have polynomial algorithms (Fortunately).

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

Three Classes of Problems

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

Undecidable, Exponential, and Polynomial.

Theoretically, all problems belong to exactly one of these three classes.

Unknown Complexity

Practically, there are a lot of problems (maybe, most) that have not been proven to be in any of the classes (Yet, maybe never will be).

Most currently "lie between" polynomial and exponential – we know of exponential algorithms, but have been unable to prove that exponential algorithms are necessary.

Some may have polynomial algorithms, but we have not yet been clever enough to discover them.

Why do we Care?

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

That's really not much more.

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

A Word about “Size”

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

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

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

The Subtlety of “Size”

But, be careful.

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

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

P = Polynomial Time

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

Some Problems in P

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

NP = Non-Det. Poly Time

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

Co-NP

- A problem is in co-NP if its complement is in NP – this is like co-RE, wrt RE problems.
- An example is the problem to determine if a boolean expression is a tautology.
  - You can check an instance to see if it does not satisfy in polynomial time.
  - However, just because one satisfies is not enough to show all do. Counterexamples are easy; proofs seem to be hard.
- The complement of satisfiability is to determine if an expression is self contradictory.

NP-Hard

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

NP-Complete; NP-Hard

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

P = NP?

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

If P ≠ NP then all NP–C problems are exponential.

Why should P = NP?

Why should P equal NP?

- There seems to be a huge "gap" between the known problems in P and Exponential. That is, almost all known polynomial problems are no worse than n3 or n4.
- Where are the $O(n^{50})$ problems?? $O(n^{100})$? Maybe they are the ones in NP–Complete?
- It's awfully hard to envision a problem that would require n^{100}, but surely they exist?
- Some of the problems in NP–C just look like we should be able to find a polynomial solution (looks can be deceiving, though).

Why Might P ≠ NP?

Why should P not equal NP?

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

Satisfiability

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

(CNF – Conjunctive Normal Form)

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

conjunction (and-ing) of "OR clauses”

Example clause:

$$c_i = (u_4 \lor u_{35} \lor \sim u_{18} \lor u_3 \ldots \lor \sim u_6)$$

SAT

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

The Proof Idea

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

Simulating NDTM

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

Tableaus

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

- It represents a configuration history of **one branch** of our NDTM's nondeterminism.
- If the NDTM runs in n^k time, the tableau is an ($n^k \times n^k$) tableau.
  - It's big enough downward because, well, the TM runs in n^k.
  - …and rightward because the TM can only *count* to n^k.
- We assume that every configuration starts and ends with a # symbol.
- We think of our tableau as looking like this in the "beginning": the starting configuration across the top, and the other configurations blank.
  - (We quote "beginning" because SAT isn't really a stateful algorithm, but just go with it for now.)
- But we've assumed that we can "represent" alphabet symbols. How do we do that, in *SAT*?

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

Encoding the Tableau: Basics

Consider a set comprised of:

The tape alphabet

The state set

The separator character

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

Consider a cell variable:

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

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

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

Encoding the Tableau: Cells

Consider our tableau alphabet:

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

Consider a cell and corresponding variable:

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

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

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

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

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

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

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

Encoding the Tableau: The Tableau

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

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

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

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

Now repeat the clause across the tableau.

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

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

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

Encoding the Tableau: Complexity

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

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

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

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

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

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

Starting and Accepting

Starting and accepting are (comparatively) easy.

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

$S = \#q_0w_1w_2\ldots w_n$□…□# so that $|S| = n^k$

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

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

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

$$\phi_{\text{accept}} = \bigvee_{1\le i,j\le n^k} \left[x_{i,j,q_\text{A}}\right]$$

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

Starting and Accepting

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

$$\phi_{\text{accept}} = \bigvee_{1\le i,j\le n^k} \left[x_{i,j,q_{\text{A}}}\right]$$

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

So now we can say that:

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

…and…

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

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

Transitions

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

> A given 2x3 **window** is **legal** if it does not violate our machine's transition function.
>
> Given the linear sets of states and tape symbols, and the finite size of 2x3 windows, we can make a **polynomial-sized set of all legal windows**.

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

> We say that ***A* is *legal*** if it **represents a legal window**. Here we have $\mathbf{q_0}$ **a R** $\mathbf{q_1}$

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

Transitions

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

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

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

$$\phi_{\text{legal}}(i,j) = \bigvee_{\substack{A=(a_1,\ldots,a_6)\\ \text{is legal}}} \begin{bmatrix} x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge \\ x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6} \end{bmatrix}$$

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

Don't be intimidated by this formula!

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

Transitions

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

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

Let a sequence *A* = (*a*$_1$, …, *a*$_6$) be a 2x3 window. ***A* is *legal*** if it **represents a legal window.**

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

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

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

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

Windows and Configurations

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

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

> The legality of the windows that don't involve the state symbol easily ensures the legality of the configuration below them.
>
> The window top-centered on the state symbol in the upper configuration is sufficient to ensure that the state symbol in the lower configuration makes a legal move.

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

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

Windows and Configurations

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

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

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

This holds all the way down the tableau.

Then we have:

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

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

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

Pulling It Together

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

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

$$\phi_{\text{accept}} = \bigvee_{1 \le i,j \le n^k} \left[x_{i,j,q_{\text{A}}}\right]$$

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

$$\phi_{\text{NDTM}} = \left(\phi_{\text{cells}} \land \phi_{\text{start}} \land \phi_{\text{accept}} \land \phi_{\text{move}}\right)$$

We have:

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

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

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

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

All are created in polynomial time.

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

SAT is NP-Complete

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

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

This means that:

> *SAT* accepts ϕ_{NDTM} if and only if such a tableau exists…
>
> …if and only if the NDTM we are encoding into ϕ_{NDTM} accepts *w*.

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

Let's convince ourselves of that a bit more.

> By definition, any NP language has an NDTM *M* that decides it in polynomial time.

We can decide any NP language with a result from *SAT* using the following algorithm:

On input <*M*, *w*>:

> Create ϕ_{NDTM} from *M* and *w*.
>
> Run the decider for *SAT* on ϕ_{NDTM}.
>
> Accept if *SAT* accepts, reject if it rejects.

SAT is NP-complete.

NP–Complete

Within a year, Richard Karp added 22 problems to this special class.

We will focus on:

3-SAT

SubsetSum

Partition

Integer Linear Programming

Vertex Cover

Independent Set

K-Color

Multiprocessor Scheduling

SAT to 3SAT

- 3-SAT means that each clause has exactly three terms
- If one term, e.g., (p), extend to $(p \vee p \vee p)$
- If two terms, e.g., $(p \vee q)$, extend to $(p \vee q \vee p)$
- Any clause with three terms is fine
- If n > three terms, can reduce to two clauses, one with three terms and one with n-1 terms, e.g., $(p1 \vee p2 \vee \ldots \vee pn)$ to
  $(p1 \vee p2 \vee z)$ & $(p3 \vee \ldots \vee pn \vee \sim z)$, where z is a new variable. If n=4, we are done, else apply this approach again with the clause having n-1 terms

SubsetSum

$S = \{s_1, s_2, \ldots, s_n\}$

set of positive integers

and an integer B.

Question: Does S have a subset whose values sum to B?

No one knows of a polynomial algorithm.

{No one has proven there isn't one, either!!}

SubsetSum $\equiv_p$ Partition

Theorem. SAT $\leq_P$ 3SAT

Theorem. 3SAT $\leq_P$ SubsetSum

Theorem. SubsetSum $\leq_P$ Partition

Theorem. Partition $\leq_P$ SubsetSum

Therefore, not only is Satisfiability in NP–Complete, but so is 3SAT, Partition, and SubsetSum.

3SAT $\leq_p$ SubsetSum

Assuming a **3SAT expression (a + ~b + c) (~a + b + ~c)**

	a	b	c	a+~b+c	~a+b+~c
a	1	0	0	1	0
~a	1	0	0	0	1
b	0	1	0	0	1
~b	0	1	0	1	0
c	0	0	1	1	0
~c	0	0	1	0	1
C1	0	0	0	1	0
C1’	0	0	0	1	0
C2	0	0	0	0	1
C2’	0	0	0	0	1
	1	1	1	3	3

SubsetSum $\equiv_p$ Partition

- [(15, 17, 27, 11, 4, 12, 33, 5, 6, 21, 2), 57]
- A solution is 15, 17, 11, 12, 2
- Mapping to Partition is
  - (15, 17, 27, 11, 4, 12, 33, 5, 6, 21, 2, 306-57, 153+57)
  - (15, 17, 27, 11, 4, 12, 33, 5, 6, 21, 2, 249, 210)
  - (15+17+11+12+2+249) = 306
  - (27+4+33+5+6+21+210) = 306
- Going other direction map above to
  - [(15, 17, 27, 11, 4, 12, 33, 5, 6, 21, 2, 249, 210), 306]

SubsetSum$\equiv_p$Partition Details

- Partition is polynomial equivalent to SubsetSum
  - Let $i_1, i_2, .., i_n$, G be an instance of SubsetSum. This instance has answer “yes” iff
    $i_1, i_2, .., i_n$, 2*Sum($i_1, i_2, .., i_n$) – G,Sum($i_1, i_2, .., i_n$) + G
    has answer “yes” in Partition. Here we assume that G ≤ Sum($i_1, i_2, .., i_n$), for, if not, the answer is “no.”
  - Let $i_1, i_2, .., i_n$ be an instance of Partition. This instance has answer “yes” iff
    $i_1, i_2, .., i_n$, Sum($i_1, i_2, .., i_n$)/2
    has answer “yes” in SubsetSum

Integer Linear Programming

- Show for 0-1 integer linear programming by constraining solution space. Start with an instance of SAT (or 3SAT), assuming variables v1,…, vn and clauses c1,…, cm
- For each variable vi, have constraint that $0 \le vi \le 1$
- For each clause we provide a constraint that it must be satisfied (evaluate to at least 1). For example, if clause cj is $v2 \lor \sim v3 \lor v5 \lor v6$ then add the constraint $v2 + (1-v3) + v5 + v6 \ge 1$
- A solution to this set of integer linear constraints implies a solution to the instance of SAT and vice versa

Assignment # 10

1. Recast the decision problem for the Boolean expression (a+b+c)(~a+~b+~c)(a+~b+~c) as a SubsetSum problem using the construction discussed in class. Indicate what rows would need to be chosen for a solution.
2. Recast the SubsetSum problem [(17, 27, 11, 2, 7, 3, 22), 39] as a Partition Problem using the construction discussed in class. Indicate what values would need to be chosen to equal 39 for the SubsetSum problem. Indicate the partitions that evenly divide the Partition Problem you posed.
3. Recast the decision problem for the Boolean expression (a+b+c)(~a+~b+~c)(a+~b+~c) as a 0,1-Integer Linear Programming problem using the construction discussed in class. Indicate what binary (0,1) values of a, b, c and d gives rise to a solution to the Integer Linear Programming problem you posed.

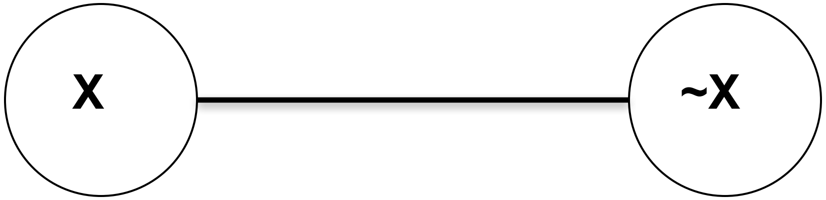
Due: November 29 at 11:59 (OPTIONAL)

VERTEX COVERING (VC) DECISION PROBLEM IS NP-HARD

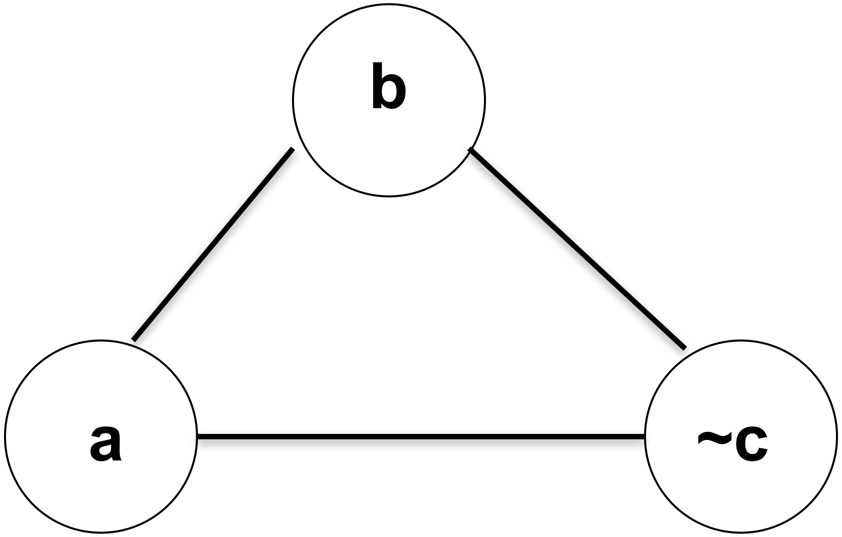
3SAT to Vertex Cover

- Vertex cover seeks a set of vertices that cover every edge in some graph
- Let $I_{3\text{-SAT}}$ be an arbitrary instance of 3-SAT. For integers n and m,
  $U = \{u_1, u_2, \ldots, u_n\}$ and $C_i = \{z_{i1}, z_{i2}, z_{i3}\}$ for $1 \le i \le m$,
  where each z_{ij} is either a u_k or u_k' for some k.
- Construct an instance of VC as follows.
- For each i, $1 \le i \le n$, construct two vertices, u_i and u_i' with an edge between them.
- For each clause $C_i = \{z_{i1}, z_{i2}, z_{i3}\}$, $1 \le i \le m$, construct three vertices z_{i1}, z_{i2}, and z_{i3} and form a "triangle on them. Each z_{ij} is one of the Boolean variables u_k or its complement u_k'. Draw an edge between z_{ij} and the Boolean variable (whichever it is). Each z_{ij} has degree 3. Finally, set k = n+2m.
- Theorem. The given instance of 3-SAT is satisfiable if and only if the constructed instance of VC has a vertex cover with at most k vertices.

VC Variable Gadget



VC Clause Gadget

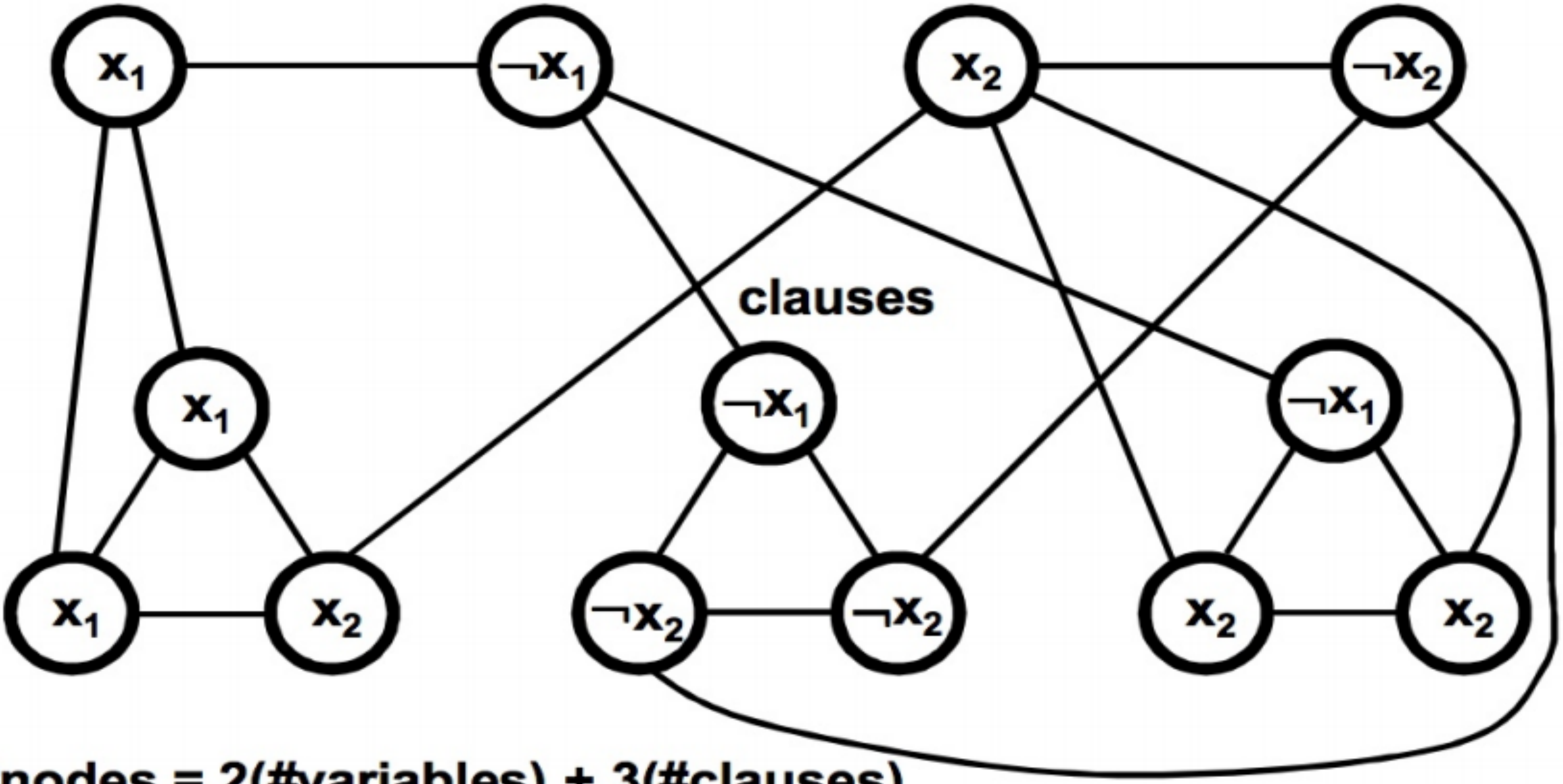


a + b + ~c

VC Gadgets Combined

$(x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$

Variables and negations of variables

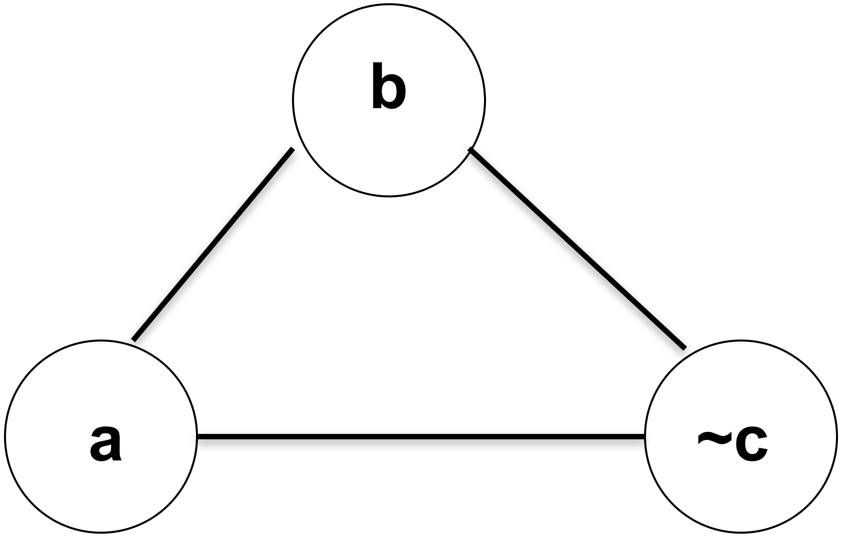


#nodes = 2(#variables) + 3(#clauses)

Independent Set

- Independent Set
  - Given Graph G = (V, E), a subset S of the vertices is independent if there are no edges between vertices in S
  - The k-IS problem is to determine for a k>0 and a graph G, whether or not G has an independent set of k nodes
- Note there is a related NP-Hard optimization problem to find a Maximum Independent Set. It is even hard to approximate a solution to the Maximum Independent Set Problem.

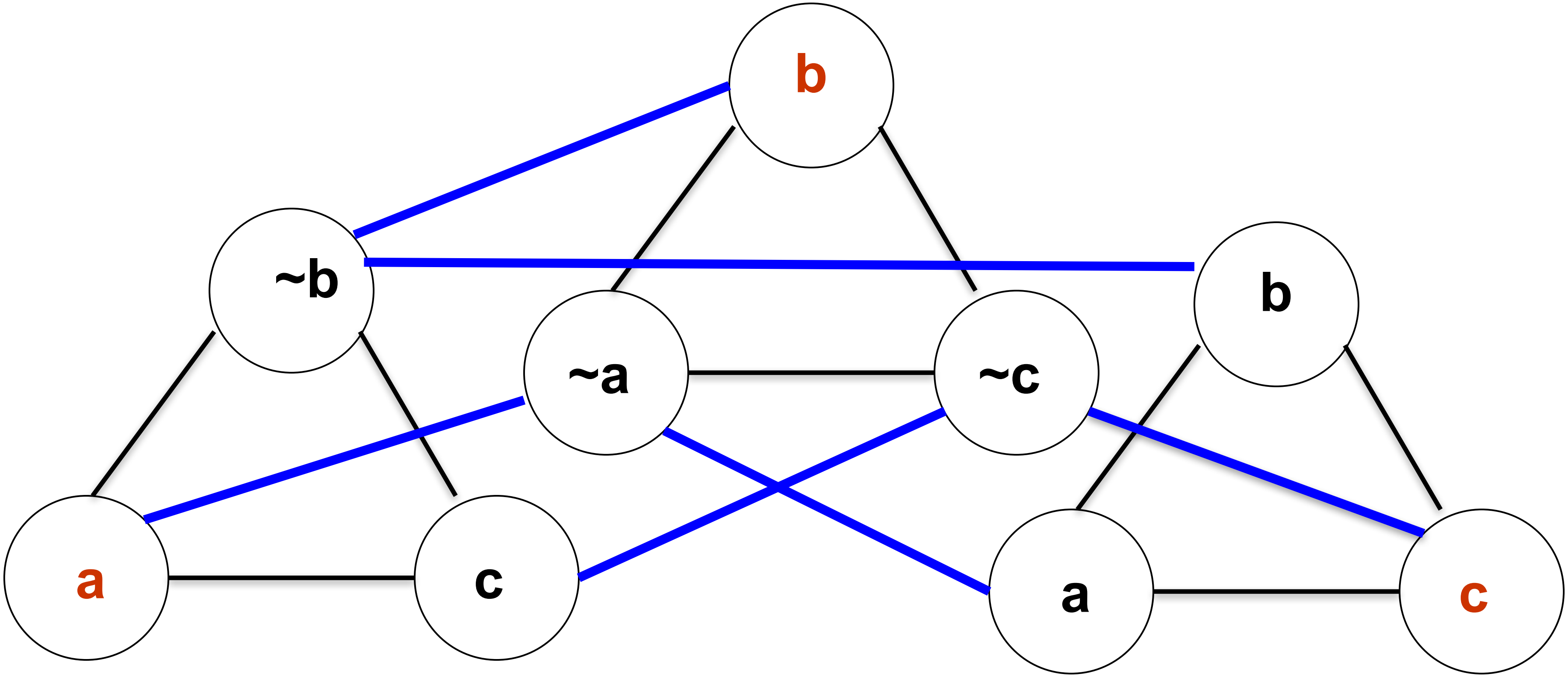
IS (VC) Clause Gadget



a + b + ~c

3SAT to IS

(a + ~b + c) (~a + b + ~c)(a + b + c), k=3
(k=number of clauses, not variables)



K-COLOR (KC) DECISION PROBLEM IS NP-HARD

K-Coloring

Given:

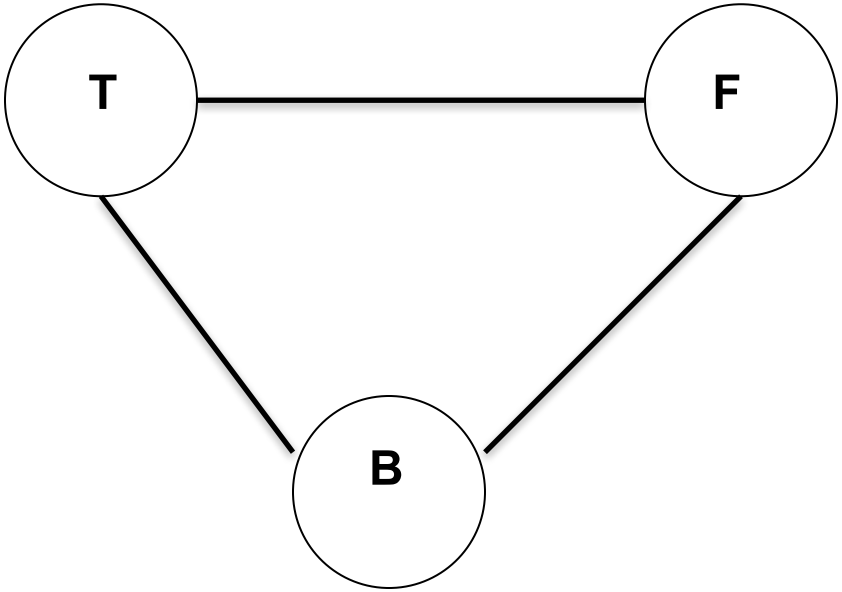
A graph G = (V, E) and an integer k.

Question:

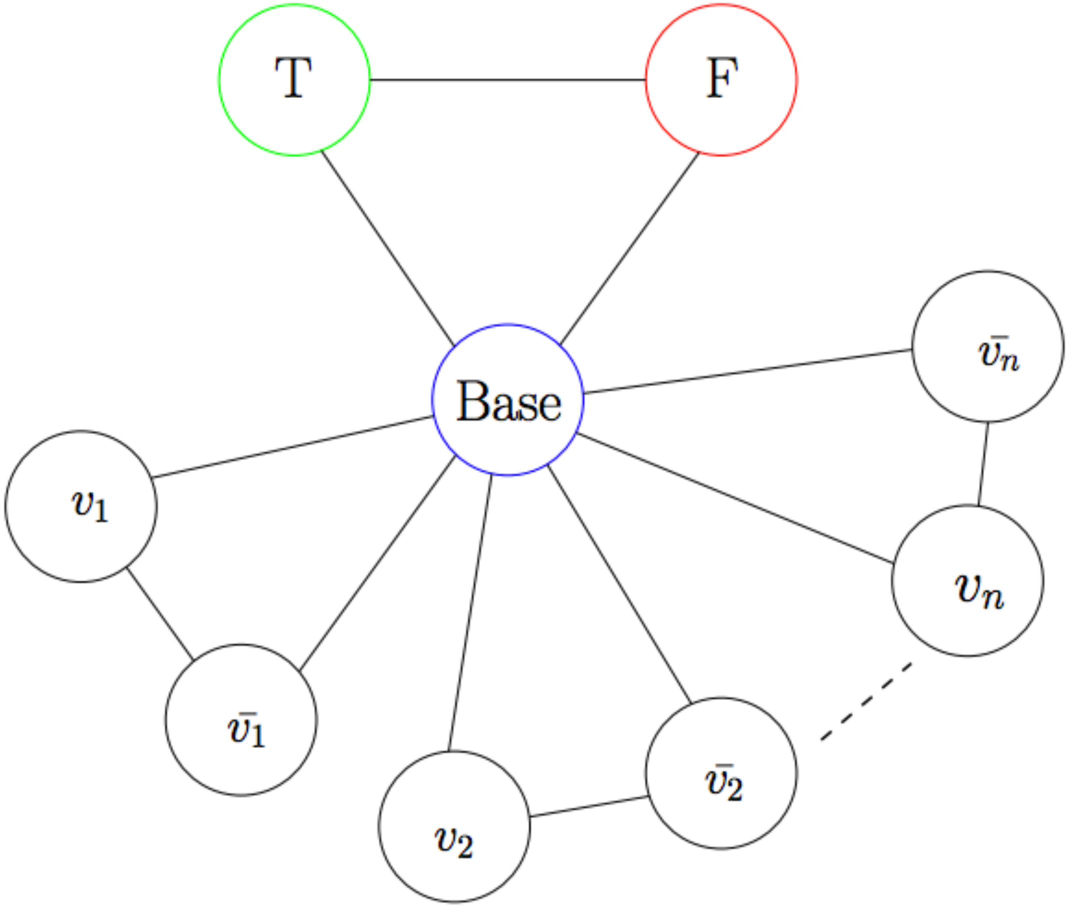
Can the vertices of G be assigned colors from a palette of size k, so that adjacent vertices have different colors and use at most k colors?

3Coloring (3C) uses k=3

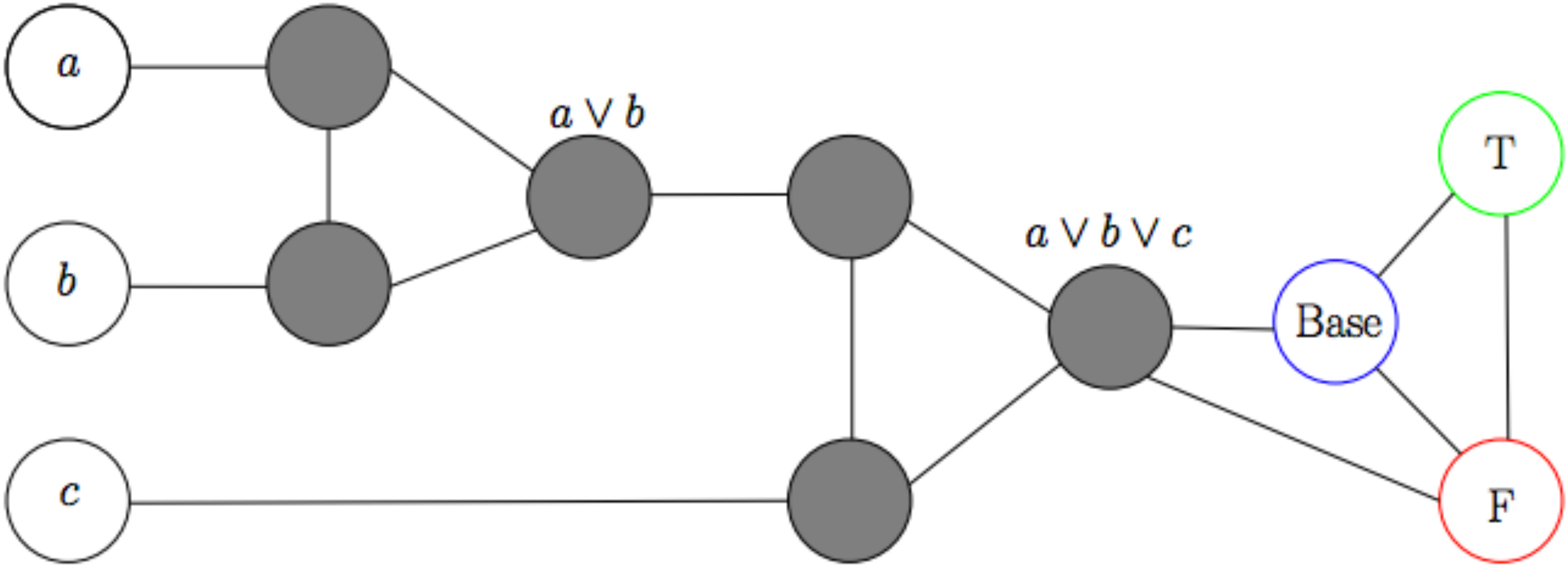
3C Super Gadget



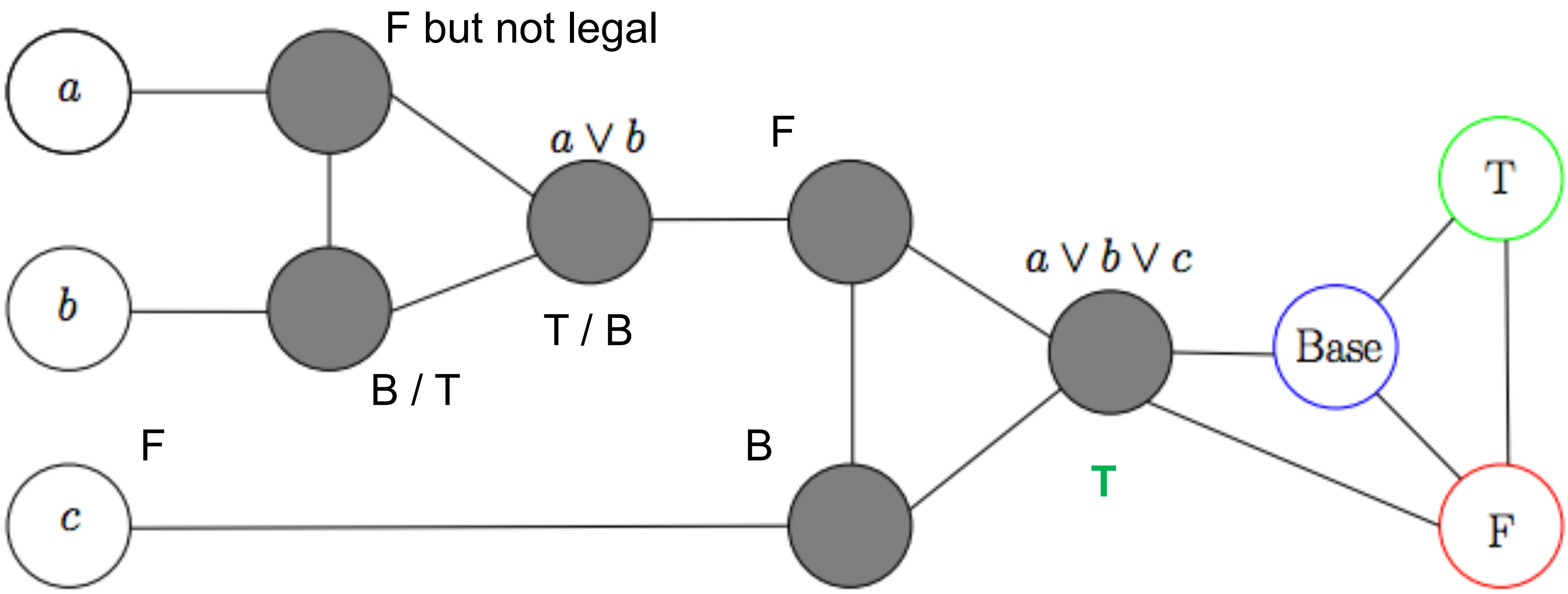
KC Super + Variables Gadget



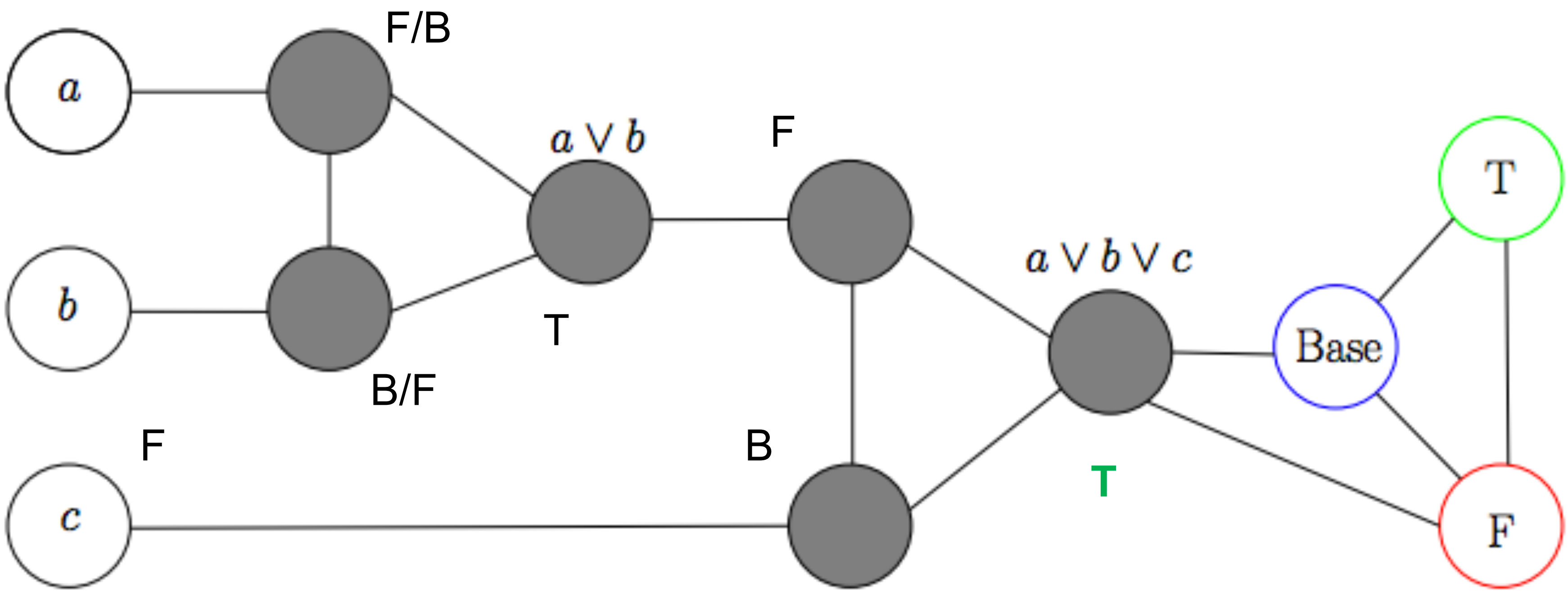
KC Clause Gadget



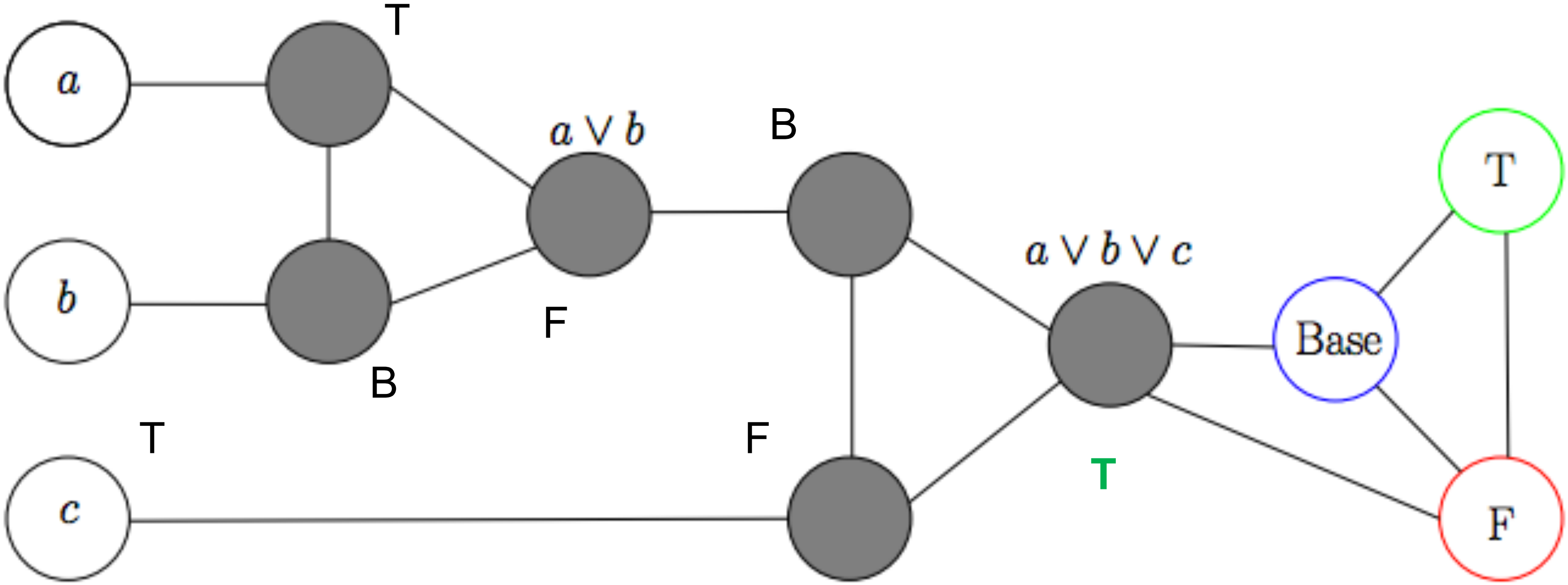
Consider ~a, ~b, ~c



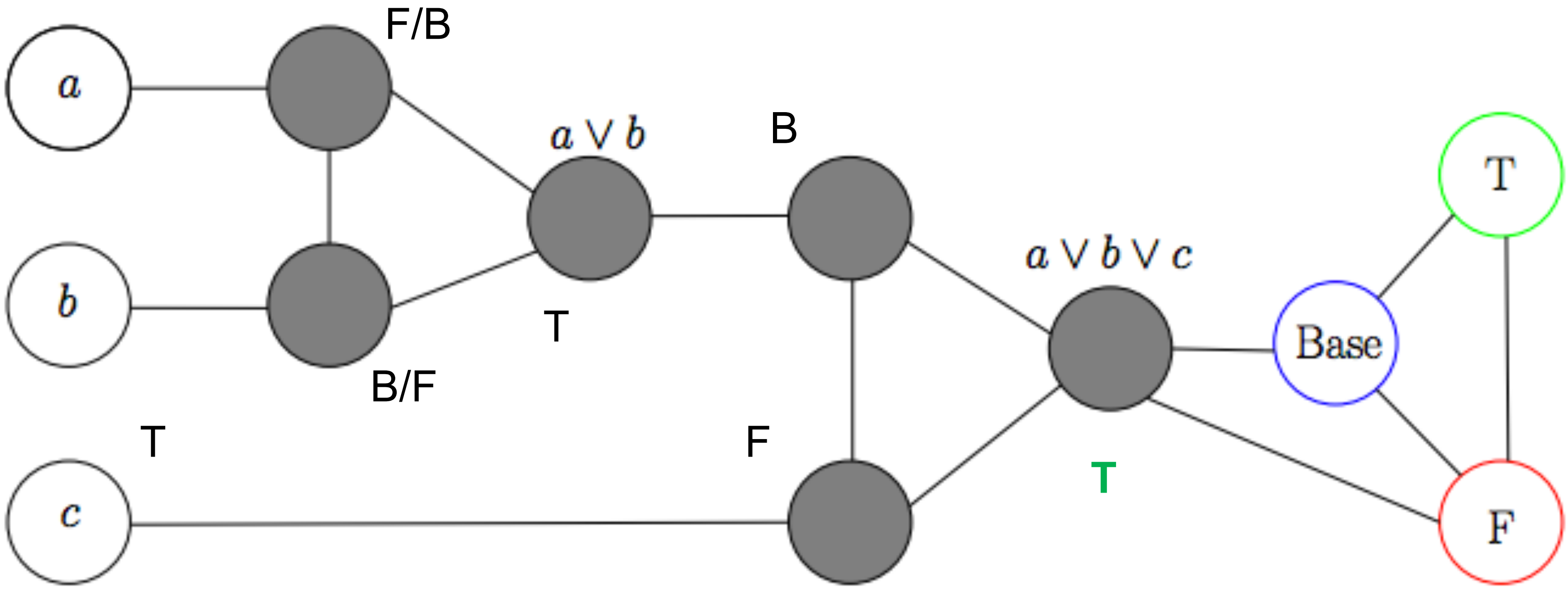
Consider a || b, ~c



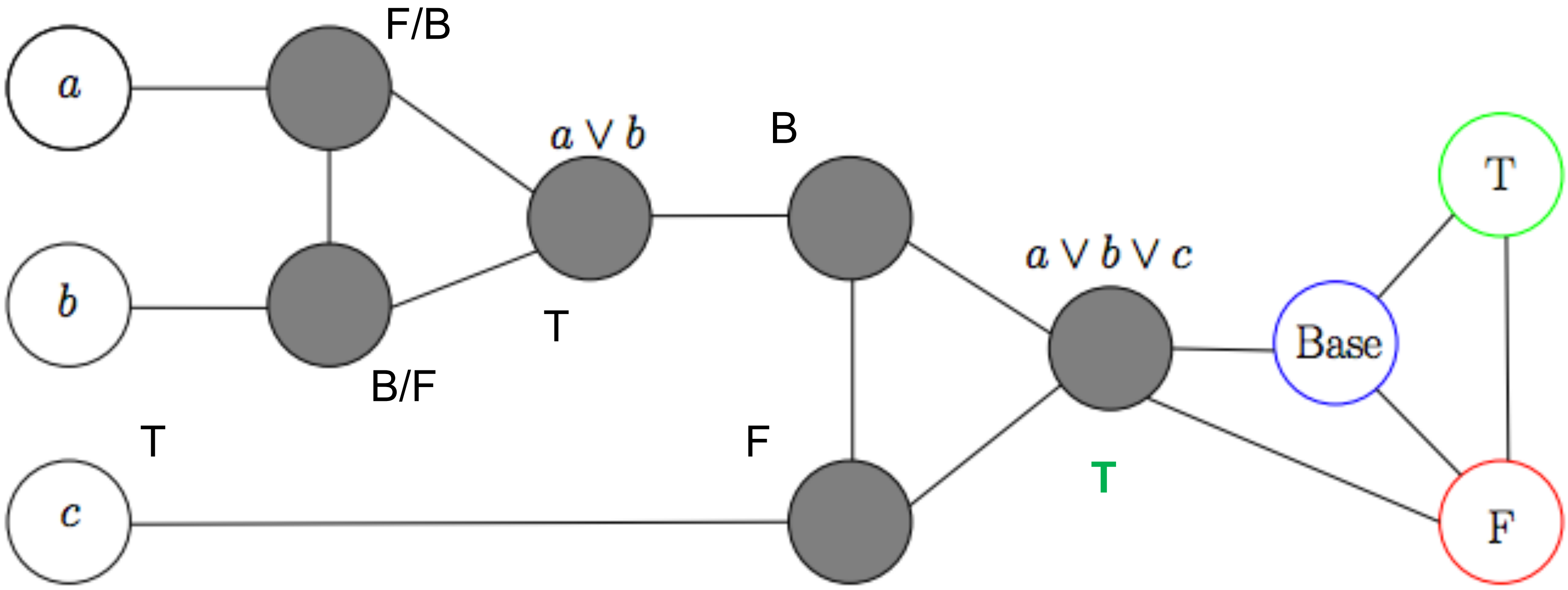
Consider ~a, ~b, c



Consider one of a || b, c

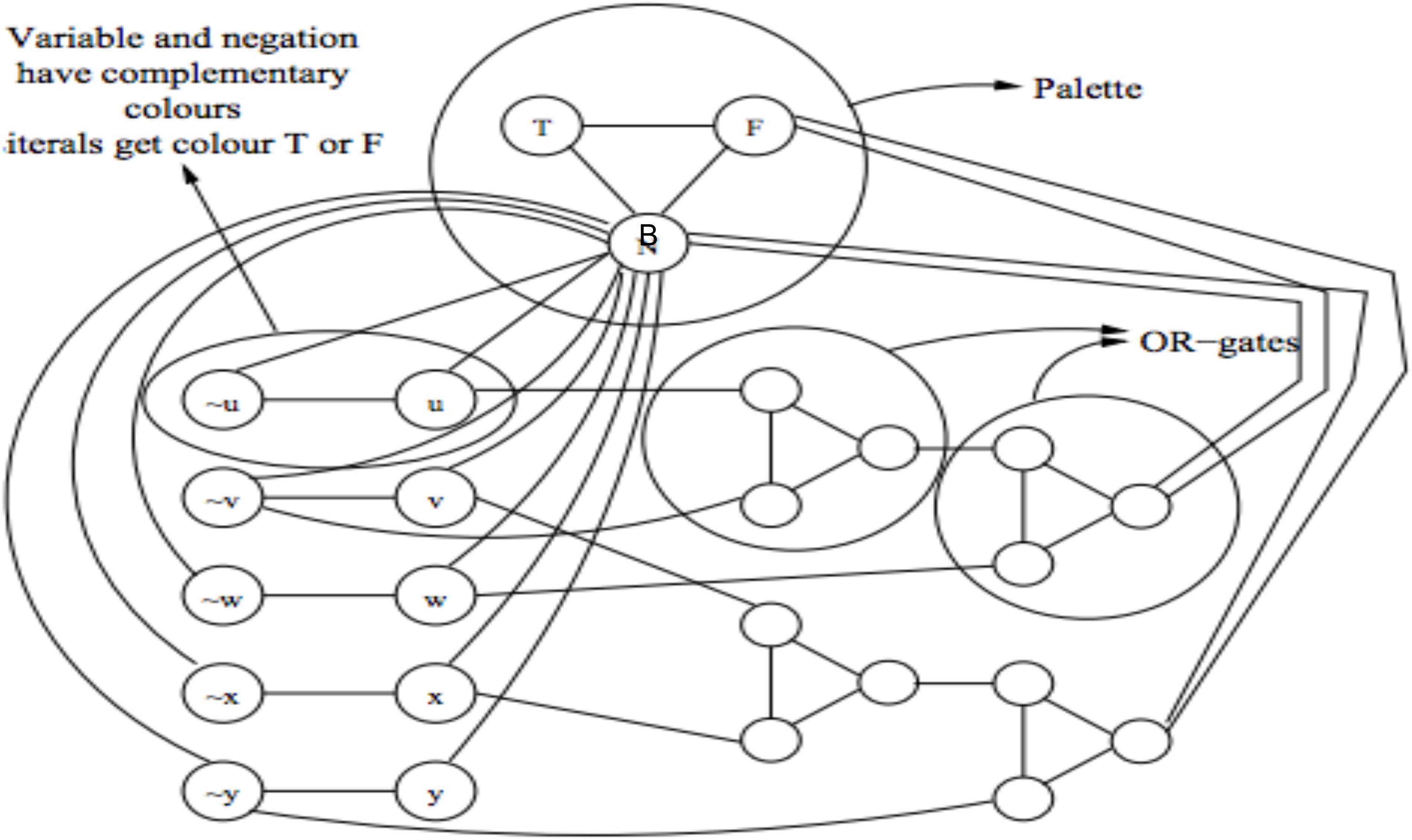


Consider a, b, c



KC Gadgets Combined

(u + ~v + w) (v + x + ~y)



K = 3

Register Allocation

- **Liveness: A variable is live if its current assignment may be used at some future point in a program's flow**
- **Optimizers often try to keep live variables in registers**
- **If two variables are simultaneously live, they need to be kept in separate registers**
- **Consider the K-coloring problem (can the nodes of a graph be colored with at most K colors under the constraint that adjacent nodes must have different colors?)**
- **Register Allocation reduces to K-coloring by mapping each variable to a node and inserting an edge between variables that are simultaneously live**
- **K-coloring reduces to Register Allocation by interpreting nodes as variables and edges as indicating concurrent liveness**
- **This is a simple mapping because it's an isomorphism**

PROCESSOR SCHEDULING IS NP-HARD

Processor Scheduling

- A Process Scheduling Problem can be described by
  - **m processors P_1, P_2, …, P_m,**
  - **processor timing functions S_1, S_2, …, S_m, each describing how the corresponding processor responds to an execution profile,**
  - **additional resources R_1, R_2, …, R_k, e.g., memory**
  - **transmission cost matrix C_{ij} ($1 \le i , j \le m$), based on proc. data sharing,**
  - **tasks to be executed T_1, T_2, …, T_n,**
  - **task execution profiles A_1, A_2, …, A_n,**
  - **a partial order defined on the tasks such that $T_i < T_j$ means that T_i must complete before T_j can start execution,**
  - **communication matrix D_{ij} ($1 \le i , j \le n$); D_{ij} can be non-zero only if $T_i < T_j$,**
  - **weights W_1, W_2, …, W_n -- cost of deferring execution of task.**

Complexity Overview

- The intent of a scheduling algorithm is to minimize the sum of the weighted completion times of all tasks, while obeying the constraints of the task system. Weights can be made large to impose deadlines.
- The general scheduling problem is quite complex, but even simpler instances, where the processors are uniform, there are no additional resources, there is no data transmission, the execution profile is just processor time and the weights are uniform, are very hard.
- In fact, if we just specify the time to complete each task and we have no partial ordering, then finding an optimal schedule on two processors is an NP-complete problem. It is essentially the subset-sum problem.

2 Processor Scheduling

The problem of optimally scheduling n tasks T_1, T_2, …, T_n onto 2 processors with an empty partial order < is the same as that of dividing a set of positive whole numbers into two subsets, such that the numbers are as close to evenly divided. So, for example, given the numbers

3, 2, 4, 1

we could try a "greedy" approach as follows:

put 3 in set 1

put 2 in set 2

put 4 in set 2 (total is now 6)

put 1 in set 1 (total is now 4)

This is not the best solution. A better option is to put 3 and 2 in one set and 4 and 1 in the other. Such a solution would have been attained if we did a greedy solution on a sorted version of the original numbers. In general, however, sorting doesn't work.

2 Processor Nastiness

Try the unsorted list

7, 7, 6, 6, 5, 4, 4, 5, 4

Greedy (Always in one that is least used)

7, 6, 5, 5 = 23

7, 6, 4, 4, 4 = 25

Optimal

7, 6, 6, 5 = 24

7, 4, 4, 4, 5 = 24

Sort it

7, 7, 6, 6, 5, 5, 4, 4, 4

7, 6, 5, 4, 4 = 26

7, 6, 5, 4 = 22

Even worse than greedy unsorted !!

Heuristics

While it is not known whether or not P = NP?, it is clear that we need to "solve" problems that are NP-complete since many practical scheduling and networking problems are in this class. For this reason we often choose to find good "<u>heuristics</u>" which are fast and provide acceptable, though not perfect, answers. The First Fit and Best Fit algorithms we previously discussed are examples of such acceptable, imperfect solutions.

Challenge Problem

Consider the simple scheduling problem where we have a set of independent tasks running on a fixed number of processors, and we wish to minimize finishing time.

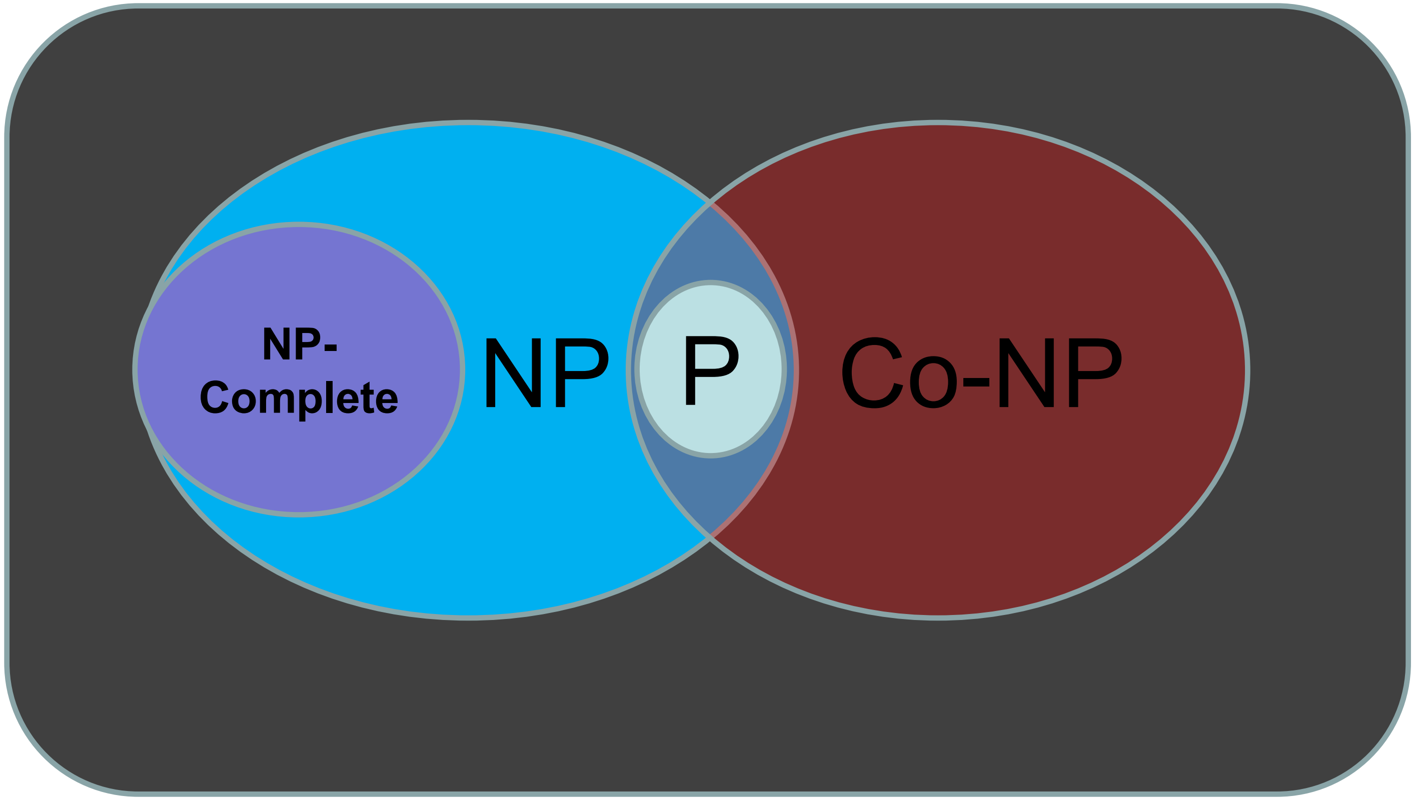
How would a list (first fit, no preemption) strategy schedule tasks with the following IDs and execution times onto four processors? Answer using Gantt chart.

(T1,4) (T2,1) (T3,3) (T4,6) (T5,2) (T6,1) (T7,4) (T8,5) (T9,7) (T10,3) (T11,4) **(2-1/m)**

Now show what would happen if the times were sorted non-decreasing. **(2-1/m)**

Now show what would happen if the times were sorted non-increasing. **(4/3-1/3m)**

UNIVERSE OF SETS



Final Exam Topics 1

- Regular languages
  - Decision Problems
    - Membership
    - Emptiness
    - Finiteness
    - Σ^*
    - Equality
    - Containment
  - Closure
    - Union/Concatenation/Star
    - Complement
    - Substitution/Quotient, Prefix, Infix, Suffix
    - Max/Min

Final Exam Topics 2

- Context free languages
  - Writing a simple CFG
  - Decision Problems
    - Membership
    - Emptiness
    - Finiteness
    - Σ* (undecidable)
    - Equality (undecidable)
    - Containment (undecidable)
  - Closure
    - Union/Concatenation/Star
    - Intersection with Regular
    - Substitution/Quotient with Regular, Prefix, Infix, Suffix
  - Non-closure
    - intersection, complement, quotient, Max/Min
  - Pumping Lemma for CFLs

Final Exam Topics 3

- Chomsky Hierarchy
  (Red involve no constructive questions)
  - Regular, CFG, CSG, PSG (type 3 to type 0)
  - FSAs, PDAs, LBAs, Turing machines
  - Length preservation or increase makes membership in associated languages decidable for all but PSGs
  - CFLs can be inherently ambiguous but that does not mean a language that has an ambiguous grammar is automatically inherently ambiguous

Final Exam Topics 4

- Computability Theory
  - Decision problems: solvable (decidable, recursive), semi-decidable (recognizable, recursively enumerable/re, generable), non-re
  - A set is re iff it is semi-decidable
  - If set is re and complement is also re, set is recursive (decidable)
  - Halting problem (K_0): diagonalization proof of undecidability
    - Set K_0 is re but complement is not
  - Set K = { f | f(f) converges }
  - Algorithms (Total): diagonalization proof of non-re
  - Reducibility to show certain problems are not decidable or even non-re
  - K and K_0 are re-complete – reducibility to show these results
  - Rice's Theorem: All non-trivial I/O properties of functions are undecidable (weak and strong versions)
  - Use of quantification to discover upper bound on complexity

Final Exam Topics 5

- Computability Applied to Formal Grammars
  (Red only results not constructions that lead to these)
  - Post Correspondence problem (PCP)
    - Definition
    - Undecidability (proof was only sketched and is not part of this course)
    - Application to ambiguity and non-emptiness of intersections of CFLs and to non-emptiness of CSLs
  - Traces of Turing computations
    - Not CFLs
    - Single steps are CFLs (use reversal of second configuration)
    - Intersections of pairwise correct traces are traces
    - Complement of traces (including terminating traces) are CFL
    - Use to show cannot decide if CFL, L, is Σ^*
    - $L = \Sigma^*$ and $L = L^2$ are undecidable for CFLs
  - PSG can mimic TM, so generate any re language; thus, membership in PSL is undecidable, as is emptiness of PSL.
  - All re sets are homomorphic images of CSLs (erase fill character)

Final Exam Topics 6

- Complexity Theory
  - Verifiers versus solvers: P versus NP
  - Definitions of NP: verify in deterministic poly time vs solve in non-deterministic polynomial time
  - Co-P and co-NP; NP-Hard versus NP-Complete
  - Basic idea behind SAT as NP-Complete
  - Reduction of SAT to 3-SAT to Subset-Sum
  - Equivalence of Subset-Sum to Partition
  - Relation of Subset-Sum and Partition to multiprocessor scheduling
  - Vertex cover, 3-coloring, register allocation, Independent set, 0-1 Integer Linear Programming
  - Gadgets for above

Supplemental Material

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 $\Rightarrow_{\boldsymbol{M}}$ such that, $\mathbf{id_i} \Rightarrow_{\boldsymbol{M}} \mathbf{id_{i+1}}$, $\mathbf{i \geq 0}$, if the execution of a single instruction of ***M*** would alter ***M***'s state from $\mathbf{id_i}$ to $\mathbf{id_{i+1}}$ or if ***M*** halts in state $\mathbf{id_i}$ and $\mathbf{id_{i+1} = id_i}$.
- $\Rightarrow^{+}_{\boldsymbol{M}}$ is the transitive closure of $\Rightarrow_{\boldsymbol{M}}$
- $\Rightarrow^{*}_{\boldsymbol{M}}$ is the reflexive transitive closure of $\Rightarrow_{\boldsymbol{M}}$

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 α**qx**β, where α (β) 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 $\mathbf{id_0}$ and that of ***M'*** is $\mathbf{id'_0}$
- Define a mapping, **h**, from id's of ***M*** into those of ***M'***, such that, $\mathbf{R}_M$ **=** { **h(d) | d** is an instance of an id of ***M*** }, and
  - $\mathbf{id'_0} \Rightarrow^*_{M'} \mathbf{h(id_0)}$, and $\mathbf{h(id_0)}$ is the only member of $\mathbf{R}_M$ in the configurations encountered in this derivation.
  - $\mathbf{h(id_i)} \Rightarrow^+_{M'} \mathbf{h(id_{i+1})}$, $\mathbf{i \geq 0}$, and $\mathbf{h(id_{i+1})}$ is the only member of $\mathbf{R}_M$ in this derivation.
- The above, in effect, provides an inductive proof that
  - $\mathbf{id_0} \Rightarrow^*_M \mathbf{id}$ implies $\mathbf{id'_0} \Rightarrow^*_{M'} \mathbf{h(id)}$, and
  - If $\mathbf{id'_0} \Rightarrow^*_{M'} \mathbf{id'}$ then either $\mathbf{id_0} \Rightarrow^*_M \mathbf{id}$, where $\mathbf{id' = h(id)}$, or $\mathbf{id'} \notin \mathbf{R_M}$

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
  TURING ≤ REGISTER ≤ FACTOR ≤
  RECURSIVE ≤ TURING
  where by **A ≤ 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 ≤ 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 $\underline{0}$ 0 0 …
- The underscore indicates the square being read. We denote this by the finite id
  1 0 1 0 0 1 1 q7 $\underline{0}$
- 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 **qi**, **L** is the number denoted by the binary sequence to the left, and **i** is the state index.
- So,
  … 0 0 1 0 1 0 0 1 1 q7 $\underline{\mathbf{0}}$ **0 0 …**
  is just (**0, 83, 7**).
  … 0 0 1 0 q5 $\underline{\mathbf{1}}$ **0 1 1 0 0 …**
  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 **r1**.
- Can also have a prolog that starts with arguments in first n registers and stores values in **r1**, **r2** and **r3** to represent Turing machines starting configuration.

Prolog

Example assuming **n** arguments (fix as needed)

```
1.       MUL_rn+1_BY_2[2] : Set rn+1 = $11…10_2$, where, #1's = r1
2.       DEC1[3,4]        : r1 will be set to 0
3.       INCn+1[1]        :
4.       MUL_rn+1_BY_2[5] : Set rn+1 = $11…1011…10_2$, where, #1's = r1, then r2
5.       DEC2[6,7]        : r2 will be set to 0
6.       INCn+1[4]        :
…
3n-2.    DECn[3n-1,3n+1]  : Set rn+1 = $11…1011…1011…1_2$, where, #1's = r1, r2,…
3n-1.    MUL_rn+1_BY_2[3n] : rn will be set to 0
3n.      INCn+1[3n-2]     :
3n+1     DECn+1[3n+2,3n+3] : Copy rn+1 to r1, rn+1 is set to 0
3n+2.    INC2[3n+1]       :
3n+3.                     : r2 = left tape, r1 = 0 (right), r3 = 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.   INC2[4]        : putting answer in r2
4.   DIV_r1_BY_2[2] : strip a 1 from r1
5.   DEC1[5,6]      : Set r1 to 0 (prepare for answer)
6.   DEC2[6,7]      : Copy r2 to r1
7.   INC1[6]        :
8.                  : Answer is now in r1
```

REGISTER $\leq$ FACTOR

Encoding a RM’s State

- This is a really easy one based on the fact that every member of $\mathbf{Z^+}$ (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 $\mathbf{p_i}$'s are distinct primes and the $\mathbf{k_i}$'s are non-zero values, except that the number **1** would be represented by $\mathbf{2^0}$.
- Let R be an arbitrary **n**-register machine, having m instructions.

  Encode the contents of registers **r1,…,rn** by the powers of $\mathbf{p_1,\ldots p_n}$.

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

  Use **pn+m+1** as prime factor that indicates simulation is done.
- This is in essence the Gödel number of the RM’s state.

Simulation by FRS

- Now, the **j**-th instruction (**1≤j≤m**) of **R** has associated factor replacement rules as follows:

j. INCr[i]

$$\mathbf{p_{n+j}x \rightarrow p_{n+i}p_r x}$$

j. DECr[s, f]

$$\mathbf{p_{n+j}p_r x \rightarrow p_{n+s}x}$$

$$\mathbf{p_{n+j}x \rightarrow p_{n+f}x}$$

- We also add the halting rule associated with **m+1** of

$$\mathbf{p_{n+m+1}x \rightarrow x}$$

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.

Example of Order

Consider the simple machine to compute **r1:=r2 – r3** (limited)

1. **DEC3[2,3]**
2. **DEC2[1,1]**
3. **DEC2[4,5]**
4. **INC1[3]**
5.

Subtraction Encoding

Start with $\mathbf{3^x5^y7}$

$7 \bullet 5\, x \rightarrow 11\, x$

$7\, x \rightarrow 13\, x$

$11 \bullet 3\, x \rightarrow 7\, x$

$11\, x \rightarrow 7\, x$

$13 \bullet 3\, x \rightarrow 17\, x$

$13\, x \rightarrow 19\, x$

$17\, x \rightarrow 13 \bullet 2\, x$

$19\, x \rightarrow x$

Analysis of Problem

- If we don't obey the ordering here, we could take an input like $\mathbf{3^5 5^2 7}$ and immediately apply the second rule (the one that mimics a failed decrement).
- We then have $\mathbf{3^5 5^2 13}$, signifying that we will mimic instruction number **3**, never having subtracted the **2** from **5**.
- Now, we mimic copying **r2** to **r1** and get $\mathbf{2^5 5^2 19}$.
- We then remove the **19** and have the wrong answer.

FACTOR $\leq$ 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$_2$), … ,(a$_n$,b$_n$))** be some factor replacement system, where **(a$_i$,b$_i$)** means that the **i**-th rule is

  a$_i$x → b$_i$x

- Encode this machine by the number **F**,

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

Simulation by Recursive # 1

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

  RULE(F, x) = μ z (1 ≤ z ≤ exp(F, 0)+1) [exp(F, 2*z-1) | x]
- Note: if **x** is divisible by **a_i**, and **i** is the least integer for which this is true, then **exp(F,2*i-1) = a_i** where **a_i** is the number of prime factors of **F** involving **p_{2i-1}**. Thus, **RULE(F,x) = i**.

  If x is not divisible by any **a_i**, **1≤i≤n**, then **x** is divisible by **1**, and **RULE(F,x)** returns **n+1**. That's why we added **p_{2n+1} p_{2n+2}**.
- Given the function **RULE(F,x)**, we can determine **NEXT(F,x)**, the number that follows **x**, when using **F**, by

  NEXT(F, x) = (x // exp(F, 2*RULE(F, x)-1)) * exp(F, 2*RULE(F, x))

Simulation by Recursive # 2

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

CONFIG(F, x, 0) = x

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

- The number of the configuration on which **F** halts is

HALT(F, x) = μ y [CONFIG(F, x, y) == CONFIG(F, x, y+1)]

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

Simulation by Recursive # 3

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

  Univ(F, x) = exp (CONFIG (F, x, HALT (F, x)), 0)

- This assumes that the answer will be returned as the exponent of the only even prime, **2**. We can fix **F** for any given Factor System that we wish to simulate.

FRS Subtraction

- $2^0 3^a 5^b \Rightarrow 2^{a-b}$
  $3*5x \rightarrow x$ 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
- $\text{RULE}(F, x) = \mu\, z\ (1 \le z \le 4)\ [\ \exp(F, 2*z-1) \mid x\ ]$
  $\text{RULE}\ (F, 3^4\ 5^2) = 1$, as 15 divides $3^4\ 5^2$
- $\text{NEXT}(F, x) = (x\ //\ \exp(F, 2*\text{RULE}(F, x)-1)) * \exp(F, 2*\text{RULE}(F, x))$
  $\text{NEXT}(F, 3^4\ 5^2) = (3^4\ 5^2\ //\ 15 * 1) = 3^3 5^1$
  $\text{NEXT}(F, 3^3\ 5^1) = (3^3\ 5^1\ //\ 15 * 1) = 3^2$
  $\text{NEXT}(F, 3^2) = (3^2\ //\ 3 * 2) = 2^1 3^1$
  $\text{NEXT}(F,\ 2^1 3^1) = (2^1 3^1\ //\ 3 * 2) = 2^2$
  $\text{NEXT}(F,\ 2^2) = (2^2\ //\ 1 * 1) = 2^2$

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)=μy[CONFIG(F,x,y)==CONFIG(F,x,y+1)] = 4
- Univ(F, x) = exp (CONFIG (F, x, HALT (F, x)), 0)
  = exp(2^2,0) = 2

Simplicity of Universal

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

  $$\mathbf{F(x) = G(\mu\ y\ H(x, y))}$$

  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, **x1, … , xn** in the form

  $\ldots 0 1^{x1} 0 1^{x2} 0 \ldots 0 1^{xn} \underline{0} \ldots$

  and ends with

  $\ldots 0 1^{x1} 0 1^{x2} 0 \ldots 0 1^{xn} 0 1^{y} \underline{0} \ldots$

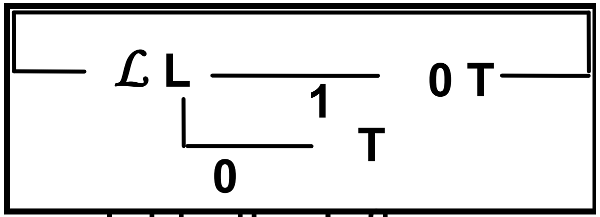
  where **y = F(x1, … , xn)**.

More Helpers

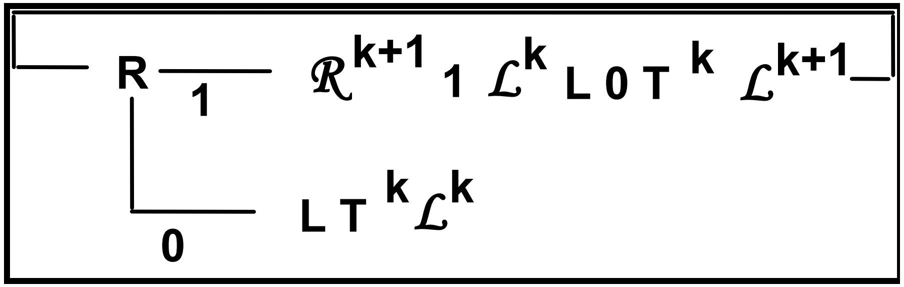
- To build our simulation we need to construct some useful submachines, in addition to the $\mathcal{R}$, $\mathcal{L}$, **R**, **L**, and $\mathbf{C_k}$ machines already defined.
- **T** -- translate moves a value left one tape square
  $\ldots\underline{?}01^x0\ldots \Rightarrow \ldots ?1^x\underline{0}0\ldots$

$\boxed{\mathbf{R1\mathcal{R}L0}}$

- Shift -- shift a rightmost value left, destroying value to its left
  $\ldots 01^{x1}01^{x2}\underline{0}\ldots \Rightarrow \ldots 01^{x2}\underline{0}\ldots$



- $\mathbf{Rot_k}$ -- Rotate a **k** value sequence one slot to the left
  $\ldots\underline{0}1^{x1}01^{x2}0\ldots 01^{xk}0\ldots$
  $\Rightarrow \ldots\underline{0}1^{x2}0\ldots 01^{xk}01^{x1}0\ldots$



R
1
$\mathcal{R}^{k+1}$ 1 $\mathcal{L}^{k}$ L 0 T^{k} $\mathcal{L}^{k+1}$
0
L T^{k} $\mathcal{L}^{k}$

Basic Functions

All Basis Recursive Functions are Turing computable:

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

$(R1)^a R$

- $I_i^n(x_1,\ldots,x_n) = x_i$

C_{n-i+1}

- $S(x) = x+1$

$C_1 1R$

Closure Under Composition

If **G, H_1, … , H_k** are already known to be Turing computable, then so is **F**, where

$F(x_1,…,x_n) = G(H1(x_1,…,x_n), … , Hk(x_1,…,x_n))$

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

$E\langle m\rangle(x_1,…,x_n, y_1,…,y_m) = E(x_1,…,x_n)$

This can be computed by the machine

$\mathcal{L}^{n+m}\ (Rot_{n+m})^n\ \mathcal{R}^{n+m}\ E\ \ \mathcal{L}^{n+m+1}\ (Rot_{n+m})^m\ \mathcal{R}^{n+m+1}$

Can now define **F** by

$H_1\ H_2\langle 1\rangle\ H_3\langle 2\rangle\ …\ H_k\langle k\text{-}1\rangle\ G\ Shift^k$

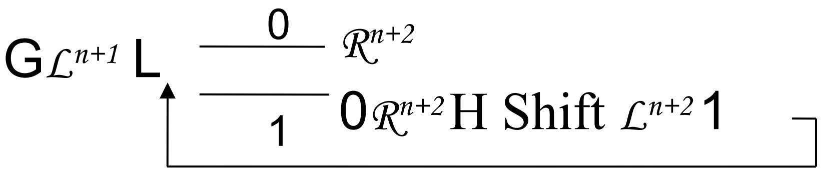
Closure Under Induction

To prove the 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:



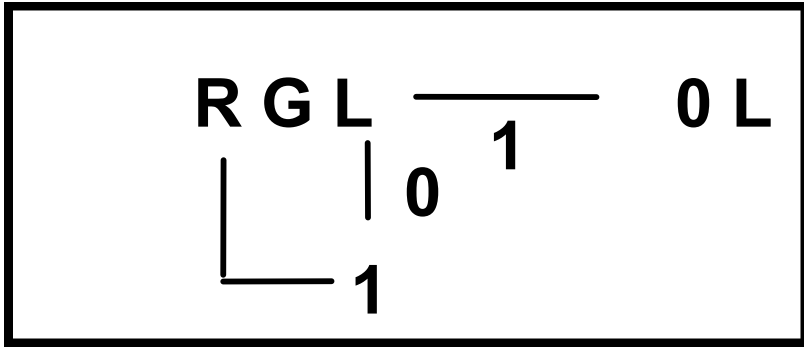
Since our Turing Machine simulator can produce the same value for any arbitrary PRF, F, we show that Turing Machines are closed under induction (primitive recursion).

Closure Under Minimization

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

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

This can be done by

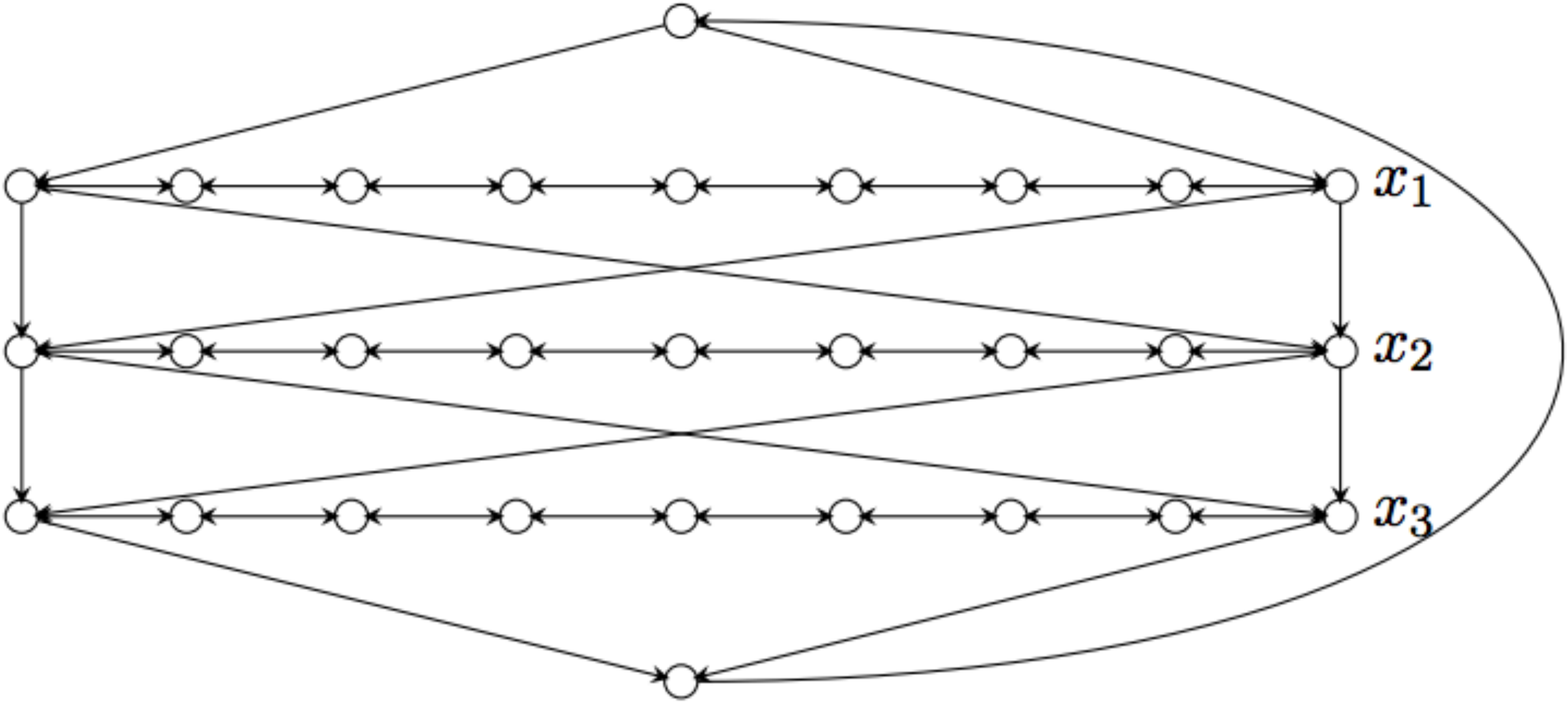


Consequences of Equivalence

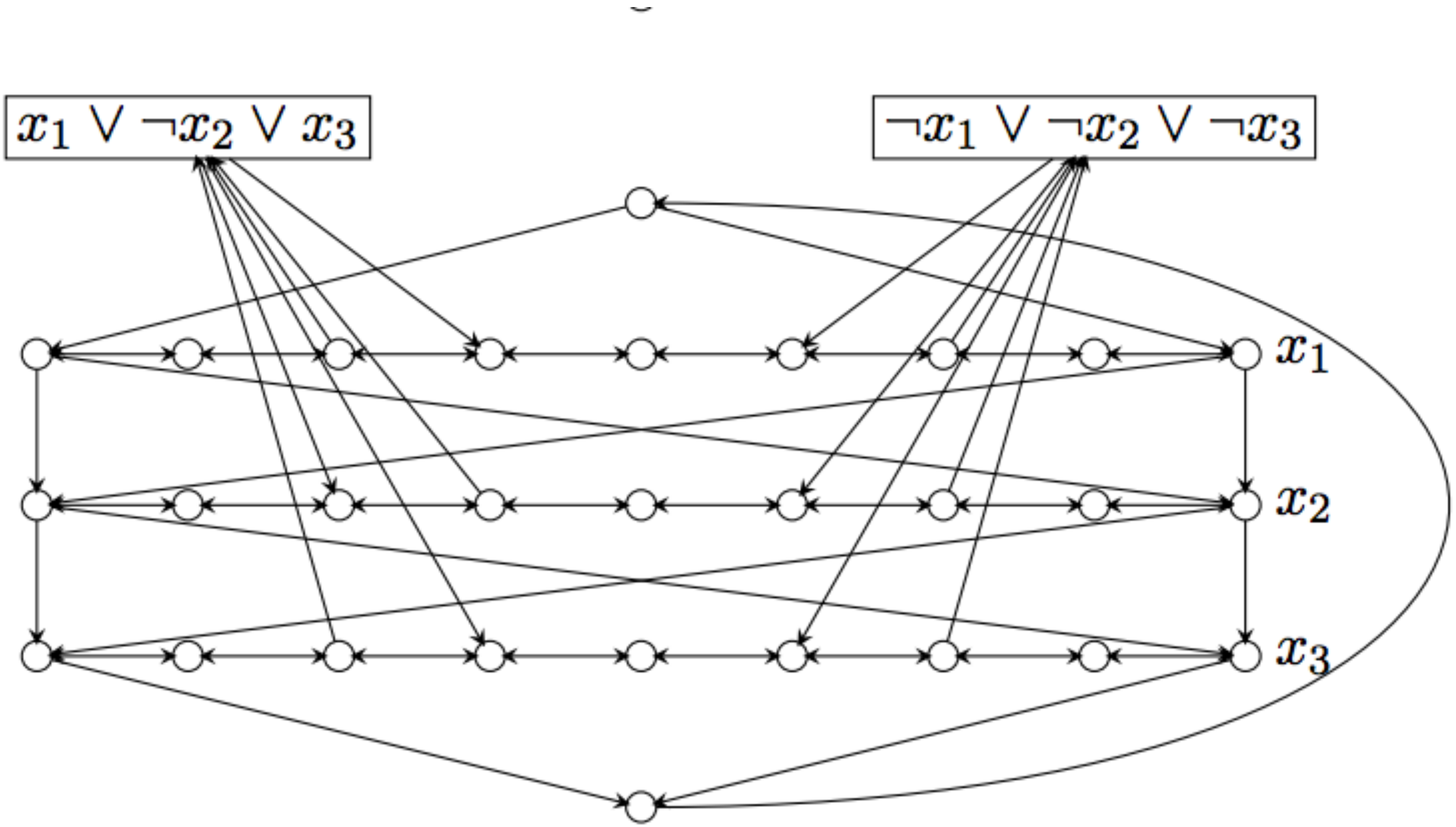
- Theorem: The computational power of Recursive Functions, Turing Machines, Register Machine, and Factor Replacement Systems are all equivalent.
- Theorem: Every Recursive Function (Turing Computable Function, etc.) can be performed with just one unbounded type of iteration.
- Theorem: Universal machines can be constructed for each of our formal models of computation.

HAMILTONIAN CIRCUIT (HC) DECISION PROBLEM IS NP-HARD

HC Variable Gadget



HC Gadgets Combined


$x_1 \vee \neg x_2 \vee x_3$
$\neg x_1 \vee \neg x_2 \vee \neg x_3$
x_1
x_2
x_3

Hamiltonian Path

- Note we can split an arbitrary node, v, into two (v’,v’’ – one, v’, has in-edges of v, other, v’’, has out-edges. Path (not cycle) must start at v’’ and end at v’ and goal is still K.

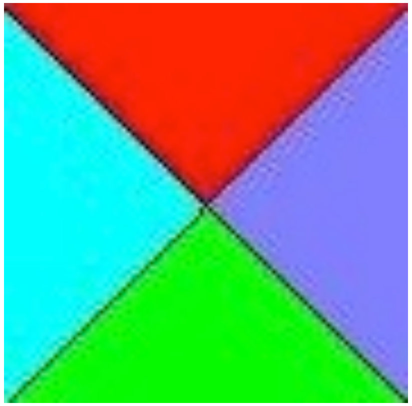
Travelling Salesman

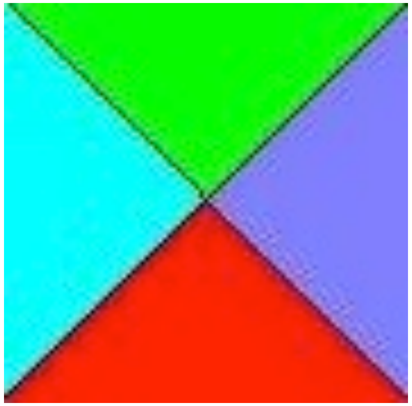
- Start with HC = (V,E), K=|V|
- Set edges from HC instance to 1
- Add edges between pairs that lack such edges and make those weights 2 (often people make these K+1); this means that the reverse of unidirectional links also get weight 2
- Goal weight is K for cycle

Tiling

Undecidable and NP-Complete Variants

Basic Idea of Tiling



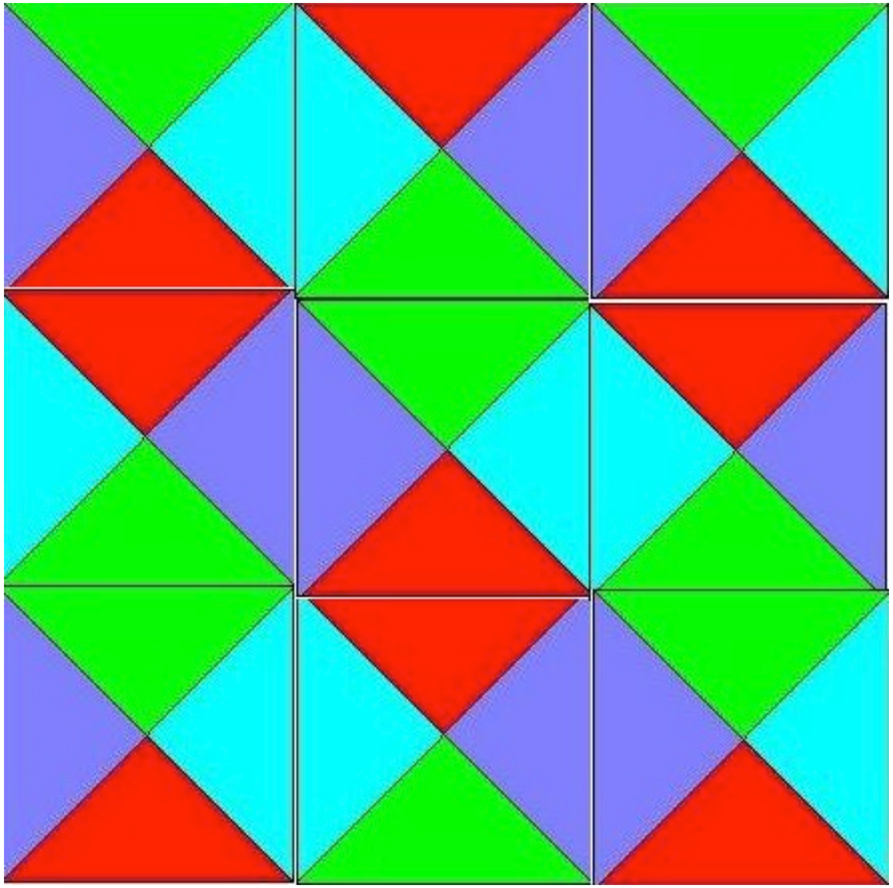


A single tile has colors on all four sides. Tiles are often called dominoes as assembling them follows the rules of placing dominoes. That is, the color (or number) of a side must match that of its adjacent tile, e.g., tile, t2, to right of a tile, t1, must have same color on Its left as is on the right side of t1. This constraint applies to top and as well as sides. Boundary tiles do not have constraints on their sides that touch the boundaries.

Instance of Tiling Problem

- A finite set of tile types (a type is determined by the colors of its edges)
- Some 2d area (finite or infinite) on which the tiles are to be laid out
- An optional starting set of tiles in fixed positions
- The goal of tiling the plane following the adjacency constraints and whatever constraints are indicated by the starting configuration.

A Valid 3 by 3 Tiling of Tile Types from Previous Slide



Some Variations

- Infinite 2d plane (impossible in general)
  - Our two tile types can easily tile the 2d plane
- Finite 2d plane (hard in general)
  - Our two tile types can easily tile any finite 2d plane
  - This is called the Bounded Tiling Problem.

- One dimensional space (hmm?)
- Infinite 3d space (not even semi-decidable in general)

Tiling the Plane

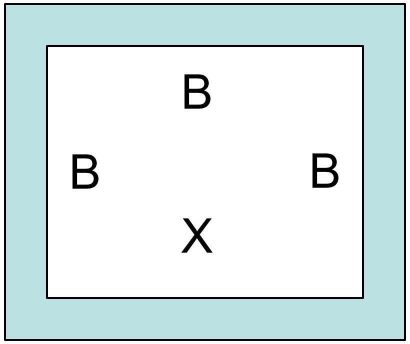
- We will start with a Post Machine, M = (Q, Σ, δ, q_0), with tape alphabet Σ = {B,1} where B is blank and δ maps pairs from Q × Σ to Q × (Σ ∪ {R,L}). M starts in state q_0
  - (Turing Machine with each action being L, R or Print)
- We will consider the case of M starting with a blank tape
- We will constrain our machine to never go to the left of its starting position (semi unbounded tape)
- We will mimic the computation steps of M
- Termination occurs if in state q reading b and δ(q,b) is not defined
- We will use the fact that halting when starting at the left end of a semi unbounded tape in its initial state with a blank tape is undecidable

The Tiling Decision Problem

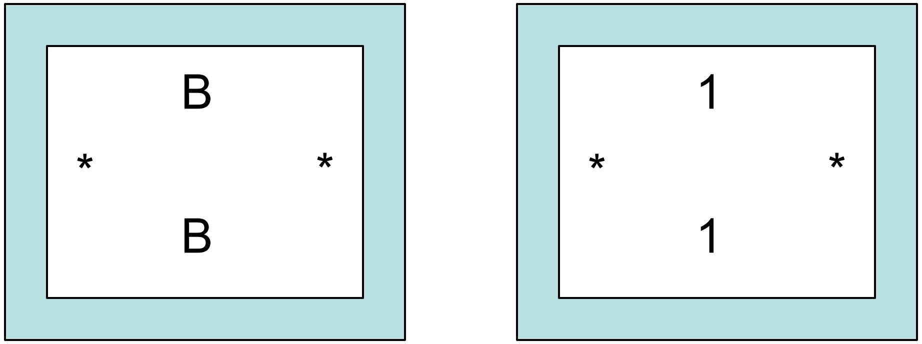
- Given a finite set of tile types and a starting tile in lower left corner of 2d plane, can we tile all places in the plane?
- A place is defined by its coordinates (x,y), $x \geq 0$, $y \geq 0$
- The fixed starting tile is at (0,0)

Colors

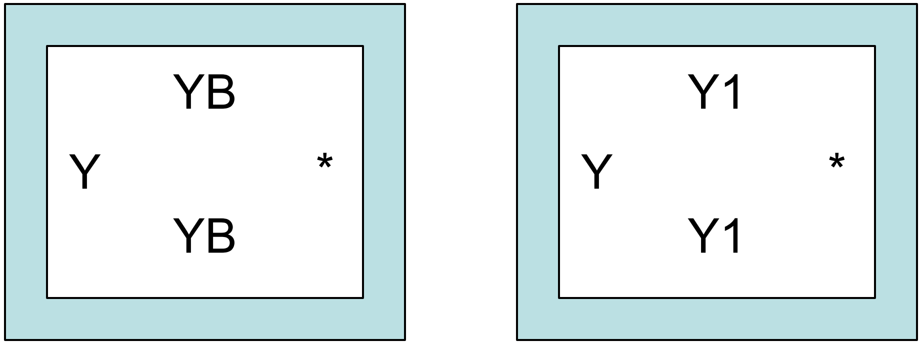
- Given M, define our tile colors as
- $\{X, Y, *, B, 1, YB, Y1\} \cup Q \times \{B,1\} \cup Q \times \{YB,Y1\} \cup Q \times \{R,L\}$
- Simplest tile (represents Blank on X axis)



Tiles for Copying Tape Cell

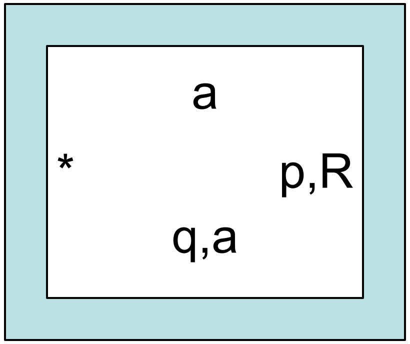


Copy cells not on left boundary and not scanned



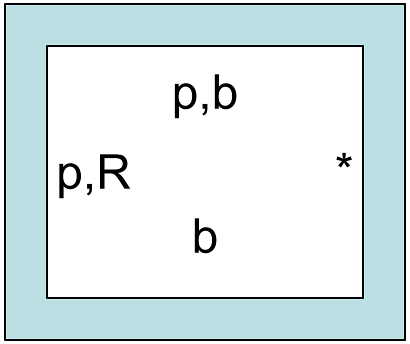
Copy cells on left boundary but not scanned

Right Move δ(q,a) = (p,R)


a

* p,R

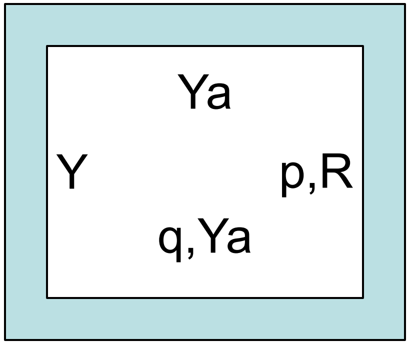
q,a


p,b

p,R *

b

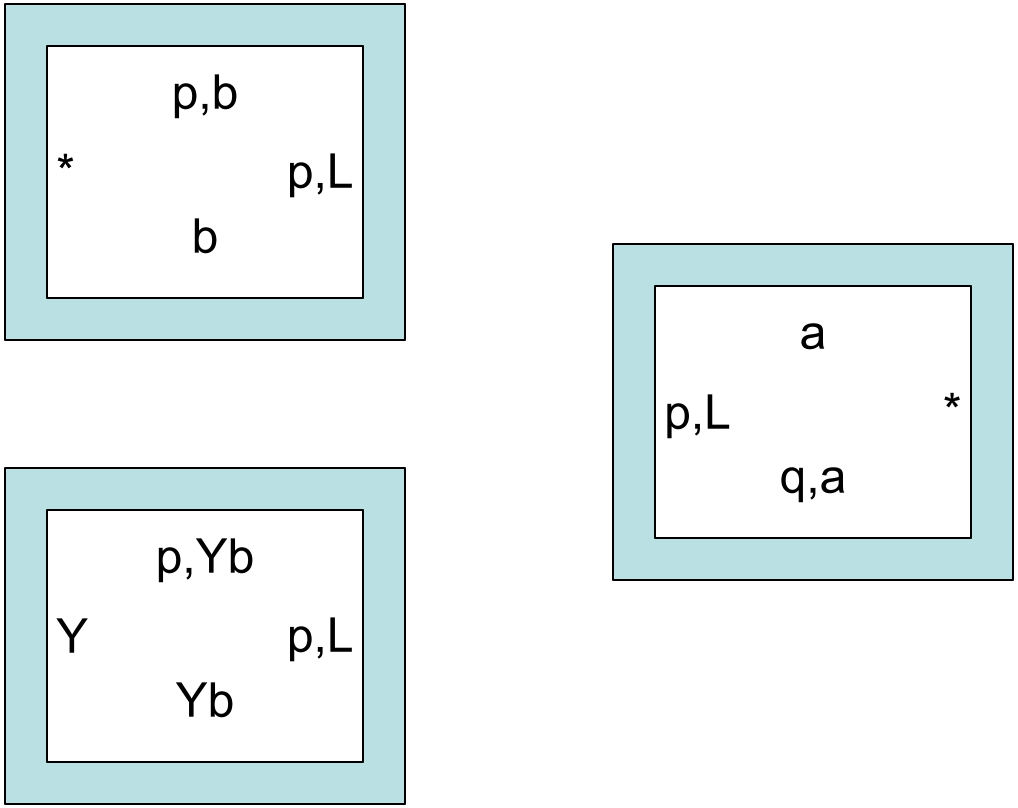
where b∈Σ


Ya

Y p,R

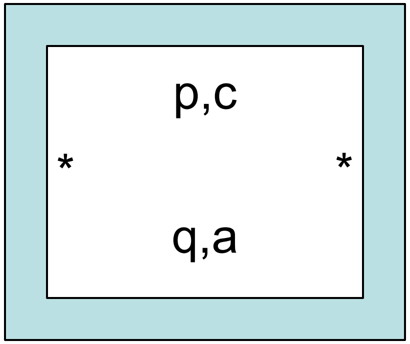
q,Ya

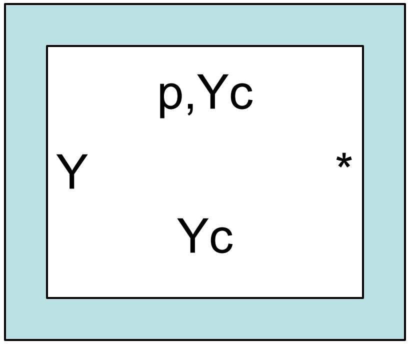
Left Move δ(q,a) = (p,L)



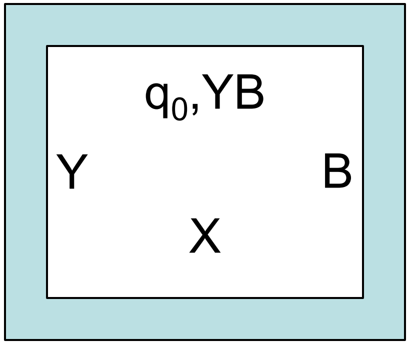
where b∈Σ

Print $\delta(q,a) = (p,c)$





Corner Tile and Bottom Row

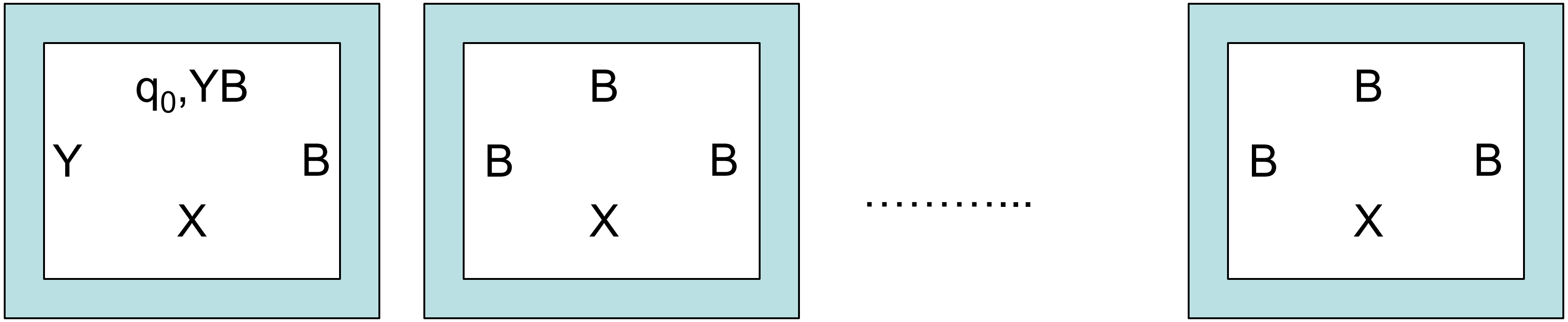


q_0,YB

Y B

X

Zero-ed Row is forced to be



q_0,YB

Y B

X

B

B B

X

…………

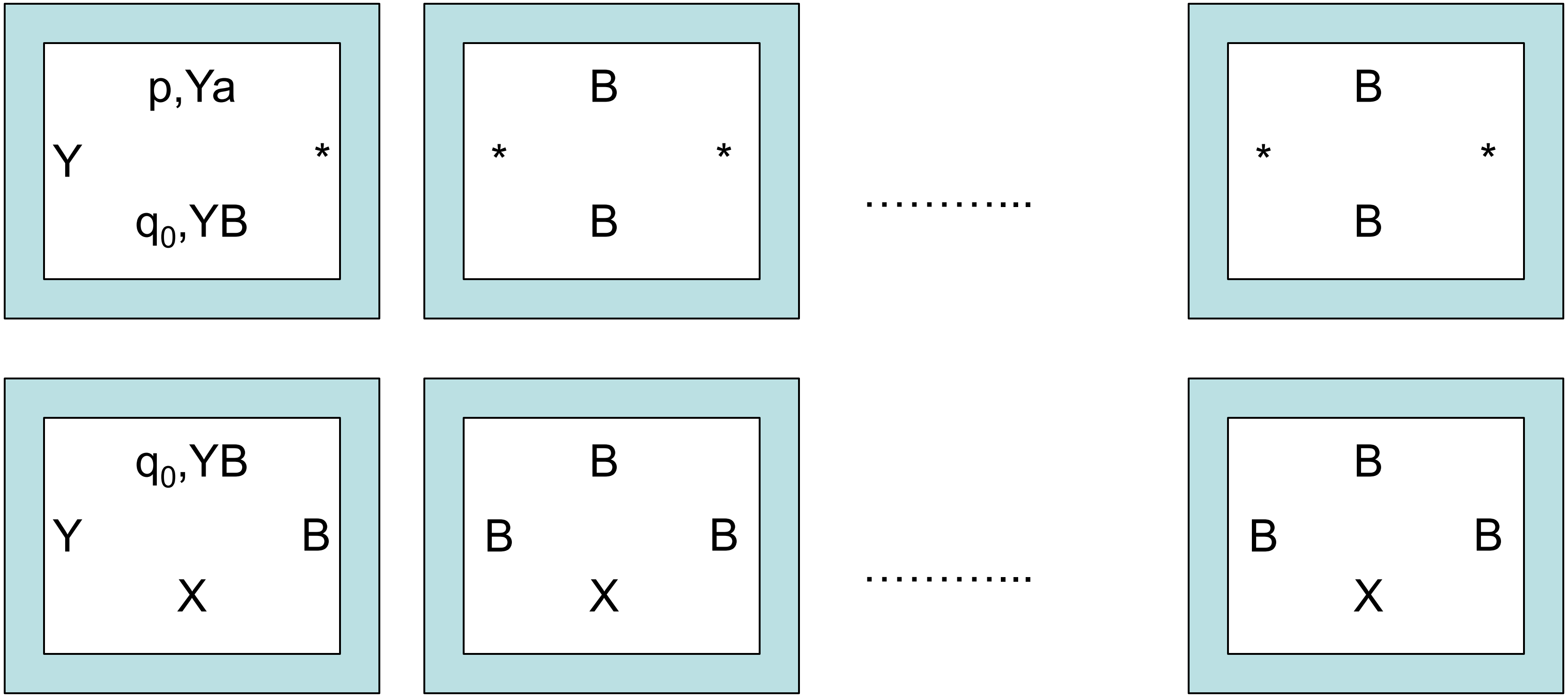
B

B B

X

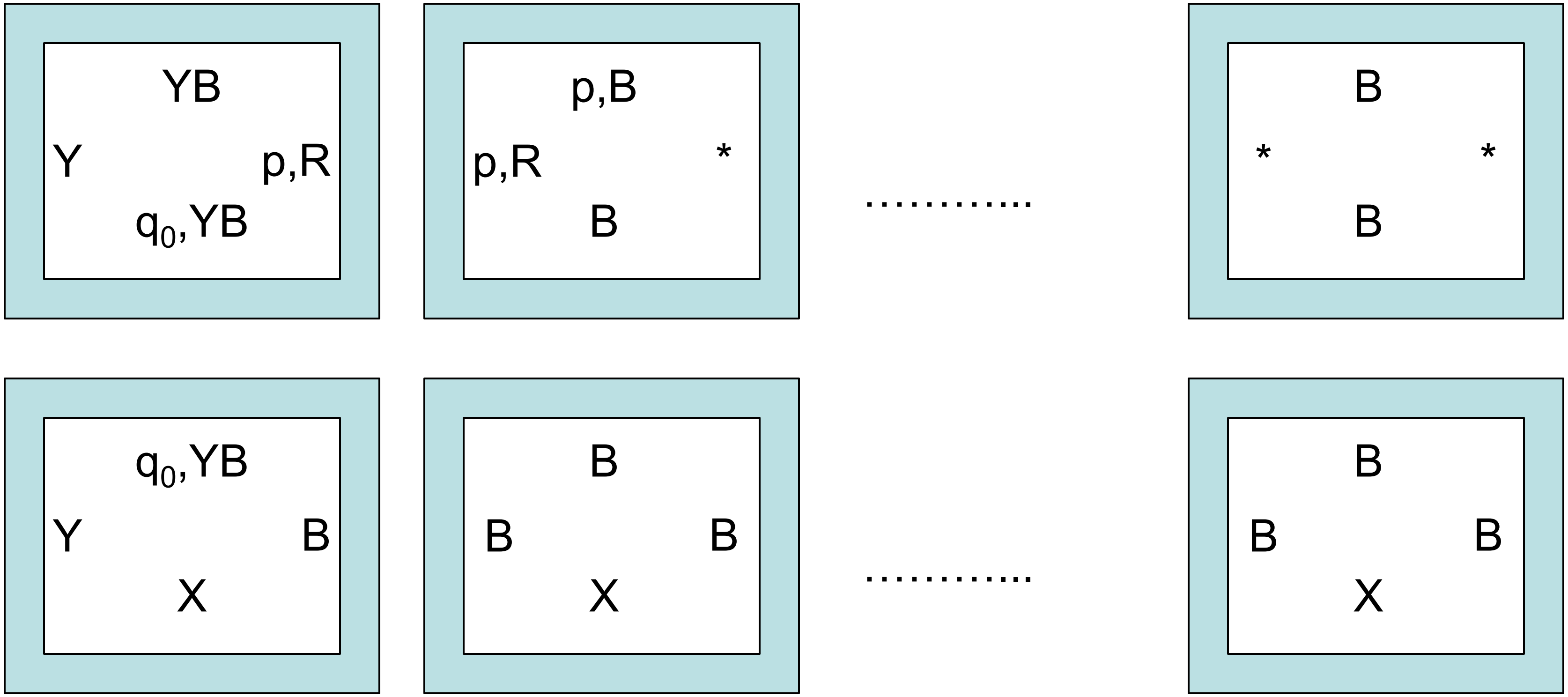
First Action Print

As we cannot move left of leftmost character first action is either right or print. Assume for now that $\delta(q_0,B) = (p,a)$



First Action Right Move

As we cannot move left of leftmost character first action is either right or print.
Assume for now that $\delta(q_0,B) = (p,R)$



The Rest of the Story Part 1

- Inductively we can show that, if the i-th row represents an infinite transcription of the Turing configuration after step i then the (i+1)-st represents such a transcription after step i+1. Since we have shown the base case, we have a successful simulation.

The Rest of the Story Part 2

- Consider the case where M eventually halts when started on a blank tape in state q_0. In this case we will reach a point where no actions fill the slots above the one representing the current state. That means that we cannot tile the plane.
- If M never halts, then we can tile the plane (in the limit).

The Rest of the Story Part 3

- The consequences of Parts 1 and 2 are that Tiling the plane is as hard as the complement of the Halting problem which is co-RE Complete.
- This is not surprising as this problem involves a universal quantification over all coordinates (x,y) in the plane.

Constraints on M

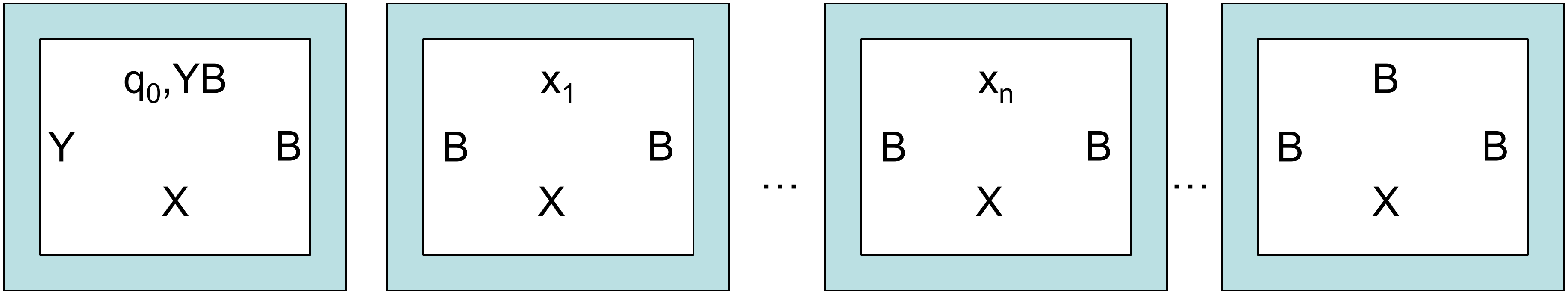
- The starting blank tape is not a real constraint as we can create M so its first actions are to write arguments on its tape.
- The semi unbounded tape is not new. If you look back at Standard Turing Computing (STC), we assumed there that we never moved left of the blank preceding our first argument.
- If you prefer to consider all computation based on the STC model then we add to M the simple prologue $(R1)^{x1}R(R1)^{x2}R\ldots(R1)^{xk}R$ so the actual computation starts with a vector of x1 … xk on the tape and with the scanned square to the blank to right of this vector. The rest of the tape is blank.
- Think about how, in the preceding pages, you could actually start the tiling in this configuration.

Bounded Tiling Problem #1

- Consider a slight change to our machine M. First, it is non-deterministic, so our transition function maps to sets.
- Second, we add two auxiliary states $\{q_a, q_r\}$, where q_a is our only accept state and q_r is our only reject state.
- We make it so the reject state has no successor states, but the accept state always transitions back to itself rewriting the scanned square unchanged.
- We also assume our machine accepts or rejects in at most n^k steps, where n is the length of its starting input which is written immediately to the right of the initial scanned square.

Bounded Tiling Problem #2

- We limit our rows and column to be of size n^k+1. We change our initial condition of the tape to start with the input to M. Thus, it looks like



- Note that there are n^k – n of these blank representations at the end. But we really only need the first.

Bounded Tiling Problem #3

- The finitely bounded Tiling Problem we just described mimics the operation of any given polynomially-bounded non-deterministic Turing machine.
- This machine can tile the finite plane of size $(n^k+1) * (n^k+1)$ just in case the initial string is accepted in n^k or fewer steps on some path.
- If the string is not accepted then we will hit a reject state on all paths and never complete tiling.
- This shows that the bounded tiling problem is NP-Hard
- Is it in NP? Yes. How? Well, we can be shown a tiling (posed solution takes space polynomial in n) and check it for completeness and consistency (this takes linear time in terms of proposed solution). Thus, we can verify the solution in time polynomial in n.

A Final Comment on Tiling

- If you look back at the unbounded version, you can see that we could have simulated a non-deterministic Turing machine there, but it would have had the problem that the plane would be tiled if any of the non-deterministic choices diverged and that is not what we desired.
- However, we need to use a non-deterministic machine for the finite case as we made this so it tiled iff some path led to acceptance. If all lead to rejection, we get stalled out on all paths as the reject state can go nowhere.

Comments on Variations

- One dimensional space (think about it)

- Infinite 3d space (really impossible in general)
  - This become a ∀ ∃ problem
  - In fact, one can mimic acceptance on all inputs here, meaning M is an algorithm iff we can tile the 3d space