Discrete II
Theory of Computation

Charles E. Hughes

COT 4210 – Fall 2014

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/Fall2014**
- Meetings: **TR 1:30PM – 2:45PM, MSB-359;**
  28 class periods, each 75 minutes long.
  Office Hours: **TR 3:15PM – 4:30PM in HEC-247C**
- GTA: **Melanie Kaprocki**
  <mskaprocki@knights.ucf.edu>
- **Please use Subject: COT4210**
  Office Hours**: TBD**

Text Material

- This and other material linked from web site.
- Text:
  - **Sipser, *Introduction to the Theory of Computation 3rd Ed.*, Course Technologies, 2013.**
- References:
  - **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: **8 to 10.**
- Exams: **Two (2) midterms and a final.**
- Material: **I will draw heavily from the text by Sipser. 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.**

Goals of Course

- Introduce Theory of Computation, including
  - Various models of computation
    - Finite State Automata and their relation to regular expressions 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
    - Undecidable problems
    - The technique of reducibility
    - The ubiquity of undecidability
  - Complexity theory
    - Order notation (this should be a review)
    - Time complexity, the sets P and NP, and the question does P=NP?

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 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 do, however, ask lots of questions in class and give lots of hints about the kinds of questions I will ask on exams. It would be a shame to miss the hints, or to fail to impress me with your insightful in-class answers.
- You are responsible for all material covered in class, whether in the 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 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.

Grading

- Grading of Assignments
  - My GTA and I will generally grade harder than our actual expectations run. Consequently, on most (not all) assignments, a grade of 90% or above will translate into a perfect grade. In general, I will award everyone ~111% of the grade they are assigned on the returned papers that are graded in this manner.

- Exam Weights
  - The weights of exams will be adjusted to your personal benefits, as I weigh exams you do well in more than those in which you do less well.

Important Dates

- Exam#1 – Thursday, September 25
- Withdraw Deadline – Mon., Oct. 27
- Exam#2 – Thursday, October 30
- Final – Thurs., Dec. 9, 1:00AM–2:50PM
- Days off: 11/11 (Veterans Day)
  11/27 (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
- Final Exam – 150 points
- Assignments – 100 points
- Bonus – best exam 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%

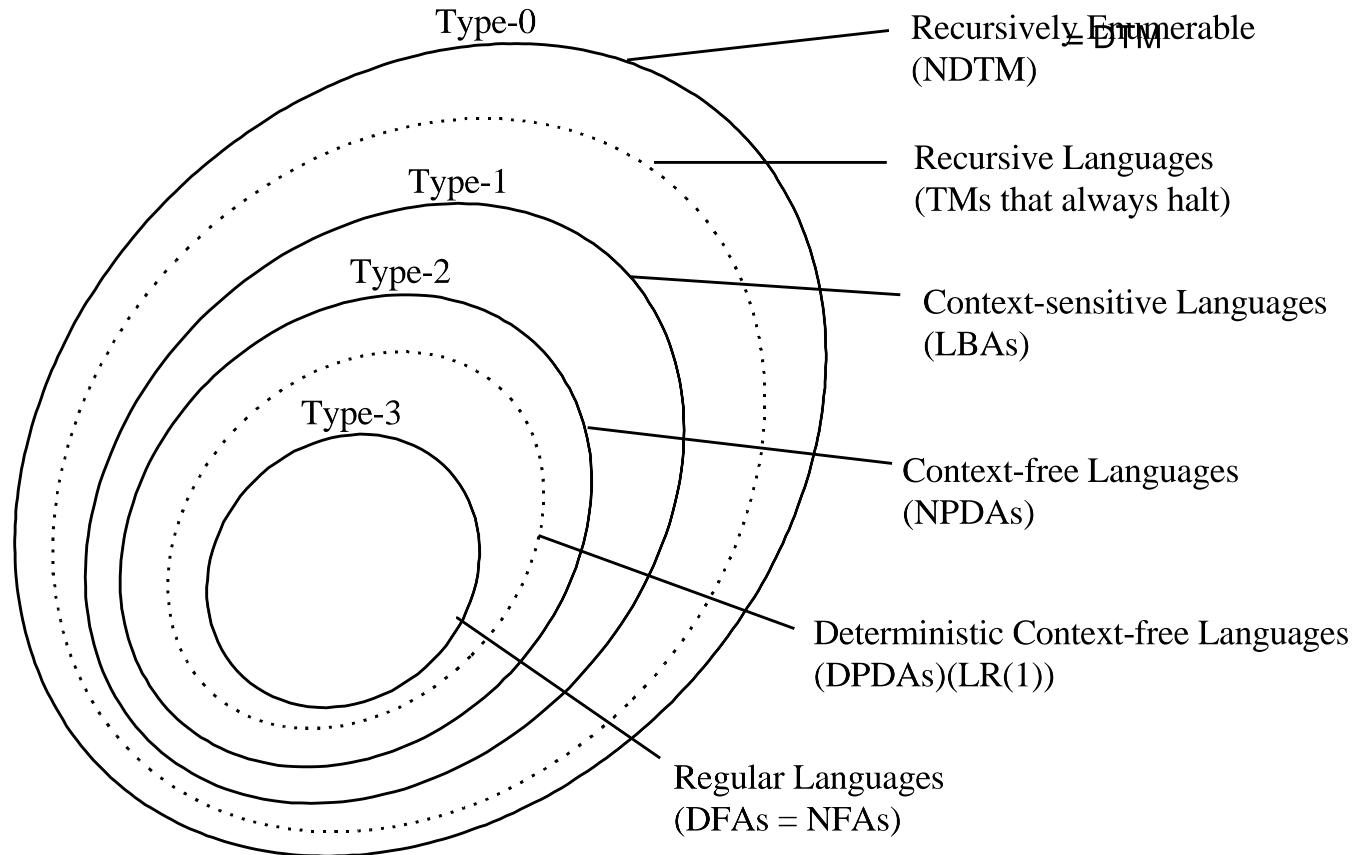
Financial Aid Related Activity

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: Melanie Kaprocki <mskaprocki@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.

Do this by late Friday, 8/22.

Forward Pass on Formal Languages and Automata



History

The Quest for Mechanizing Mathematics

Hilbert, Russell and Whitehead

- Late 1800's to early 1900's
- Axiomatic schemes
  - Axioms plus sound rules of inference
  - Much of focus on number theory
- First Order Predicate Calculus
  - $\forall x \exists y\ [y > x]$
- Second Order (Peano's Axiom)
  - $\forall P\ [[P(0)\ \&\&\ \forall x[P(x) \Rightarrow P(x+1)]] \Rightarrow \forall x P(x)]$

Hilbert

- In 1900 declared there were 23 really important problems in mathematics.
- Belief was that the solutions to these would help address math's complexity.
- Hilbert's Tenth asks for an algorithm to find the integral zeros of polynomial equations with integral coefficients. This is now known to be impossible (In 1972, Matiyacevič showed this undecidable).

Hilbert's Belief

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

Gödel

- In 1931 he showed that any first order theory that embeds elementary arithmetic is either incomplete or inconsistent.
- He did this by showing that such a first order theory cannot reason about itself. That is, there is a first order expressible proposition that cannot be either proved or disproved, or the theory is inconsistent (some proposition and its complement are both provable).
- Gödel also developed the general notion of recursive functions but made no claims about their strength.

Turing (Post, Church, Kleene)

- In 1936, each presented a formalism for computability.
  - **Turing and Post devised abstract machines and claimed these represented all mechanically computable functions.**
  - **Church developed the notion of lambda-computability from recursive functions (as previously defined by Gödel and Kleene) and claimed completeness for this model.**
- Kleene demonstrated the computational equivalence of recursively defined functions to Post-Turing machines.
- Church's notation was the lambda calculus, which later gave birth to Lisp.

More on Emil Post

- In the 1920's, starting with notation developed by Frege and others in 1880s, Post devised the truth table form we all use now for Boolean expressions (propositional logic). This was a part of his PhD thesis in which he showed the axiomatic completeness of the propositional calculus.
- In the late 1930's and the 1940's, Post devised symbol manipulation systems in the form of rewriting rules (precursors to Chomsky's grammars). He showed their equivalence to Turing machines.
- In 1940s, Post showed the complexity (undecidability) of determining what is derivable from an arbitrary set of propositional axioms.

Languages

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 $|\lambda| = 0$)
    $\forall a \in \Sigma$, $a \in \Sigma^*$ (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·x" denotes "a *concatenated to* x" and is formed by appending the symbol a to the left end of x. Similarly, x·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".

Languages

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

Operations on Strings

- Let s, t be arbitrary strings over Σ
  - $s = a_1 a_2 \ldots a_j$, $j \geq 0$, 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: $s^n = ss \ldots s$ (n times) Note: $s^0 = \lambda$
- reverse: $s^R = a_j a_{j-1} \ldots a_1$
- substring: for s, 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)
  - $L^* = L^0 \cup L^1 \cup L2 \cup \ldots Ln \ldots$ (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 $n \geq 0$.

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, can be represented as some string over a finite alphabet, Σ_P; that is, $P \in \Sigma^*$, and thus there are at most a countably infinite number of programs over Σ_P.

Computable Languages 2

7. A programming language, L_P, is an element of $\mathcal{P}(\Sigma_P^*)$; $|L_P|$ is countable.
8. Each program, P, defines a function, $F_P: \Sigma_I^* \rightarrow \Sigma_O^*$
9. F_P defines an input language P_I and an output language P_O.
10. 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.
11. But, there are an uncountable number of possible languages over any given alphabet – see 3 and 5.
12. Thus there must be languages over a given alphabet that have no description – 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.

Sets, Sequences, Relations, Functions and Infinity

Mostly compliments of Dr. Workman

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
    $S = \{ x \in \Delta \mid P(x) \}$, 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. $S = \{ x \in UCF \mid x \text{ is a CS major} \}$
- The empty set is denoted, $\varnothing$, and is the set with no members; that is,
  $\varnothing = \{ \}$. Also, the predicate, $x \in \varnothing$, is always false!

More on Sets

- If $S \neq \emptyset$, then there exists an x for which $x \in S$ is true; this predicate is read "x is an element of S" or "x is a member of S". The symbol "$\in$" denotes the member relation. $x \notin S$ is true when x is not in S.
- We use normal set operation of union $(A \cup B)$, intersection $(A \cap B)$ and complement ~A (usually A with a bar on it).
- If A and B are sets, then we write "$A \subseteq B$" to mean that A is a subset of B. This means that for all $x \in A$, $x \in B$. Or, $\forall x\, [x \in A \Rightarrow x \in B]$.
- The expression, "$A \subset B$" means that A is a proper subset of B. Mathematically, $\forall x\, [x \in A \Rightarrow x \in B]$ and $\exists y\, [\, y \in B$ and $y \notin A]$
  Note the text uses the subset notation with a line through the lower bar, but that symbol is not available in my fonts.
- The cross (Cartesian) product of two sets A and B is denoted, $A \times B$, and is the set defined as follows: $A \times B = \{ (a,b) \mid a \in A \text{ and } b \in B \}$. "(a,b)" is an expression composed from elements, a,b, selected arbitrarily from sets A and B, respectively. If $A \neq B$, then $A \times B \neq B \times A$.
  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) ≠ (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;

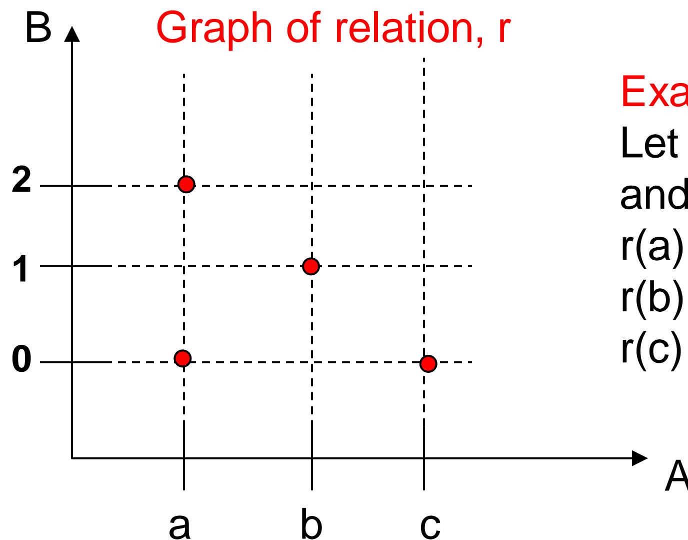
  We write, r: $A \rightarrow B$, and we mean that r assigns to every member of A a subset of B; that is, for every $a \in A$, $r(a) \subseteq B$ and $r(a) \neq \emptyset$.

  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 $b \in B$ 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 $a \in A$, $r(a) = \emptyset = \{ \}$.

Relations

- A relation can be graphed as illustrated by the example below.



Example:
Let A = { a, b, c} , B = { 0, 1, 2 },
and r = { (a,0), (a,2), (b,1), (c,0) }
r(a) = { 0, 2 }
r(b) = { 1 }
r(c) = { 0 }

Functions

- Functions are special types of relations. Specifically, a relation $f: A \rightarrow B$, is said to be a (total) function from A to B if and only if, for every $a \in A$, $f(a)$ 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 $a \in A$. In this case we write $|f(a)| \leq 1$, for every a in A; note that $|f(a)| = 0$ if and only if $f(a) = \emptyset$, 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 $x \neq y$ implies $f(x) \neq f(y)$. A total function that is one-to-one is sometimes called an *injection.*
- A function, $f: A \rightarrow B$, is said to be onto if and only if for every $y \in B$ there is an $x \in A$ 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, $\mathcal{N}$. (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.

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

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

To summarize, the cardinality of any <u>finite set</u> (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 $|S| \in \mathcal{N}$; otherwise, S is said to be infinite. A set S is said to be countable if and only if S is finite or $|S| = |\mathcal{N}|$; otherwise S is said to be uncountable.

Infinities

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

For example:

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

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

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

Cantor and Infinities

The previous "brash" statement suggests there are at least two infinite cardinals, $|\mathcal{N}|$ and $|\mathcal{R}|$. Furthermore, $|\mathcal{N}|$ is a countable cardinal and $|\mathcal{R}|$ 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 $|\mathcal{N}| = \aleph_0$, and although many mathematicians believe that $\aleph_1 = |\mathcal{R}|$, this has never been proven from the axioms of mathematical set theory.

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.

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

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

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

$, \{0,1\}, \{0,2\}, \{0,3\}, \ldots$

$, \{0,1,2\}, \ldots$

$\ldots \mathcal{N}\}$

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

- There are sets whose cardinalities are greater than $\aleph_0$. These sets are uncountably infinite, whereas those that correspond to $\mathcal{N}$ are countably infinite.
- Note that a set can be countable and yet there is no effective way to describe its correspondence with $\mathcal{N}$. 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 $\mathcal{Z}$ and $\mathcal{Q}$

1. We show that $|\mathcal{N}| = |\mathcal{Z}|$.
   $|\mathcal{N}| \leq |\mathcal{Z}|$: Define g: $\mathcal{N} \rightarrow \mathcal{Z}$ as follows: g(i) = i
   $|\mathcal{Z}| \leq |\mathcal{N}|$: Define f: $\mathcal{Z} \rightarrow \mathcal{N}$ as follows:

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

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

2. To show $|\mathcal{N}| = |\mathcal{Q}|$ we develop the proof in two steps:
   (a) Lemma – prove that $|A| \leq |S|$ for every subset A of S.
   Note: This is what we did for $|\mathcal{N}| \leq |\mathcal{Z}|$
   (b) Prove that $|\mathcal{N} \times \mathcal{N}| = |\mathcal{N}|$.

$|\text{Subset}| \leq |\text{Parent Set}|$

Lemma A. $|A| \leq |S|$, for every subset A of S.

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

$| \mathcal{N} \times \mathcal{N} | = | \mathcal{N} |$

Lemma B. $| \mathcal{N} \times \mathcal{N} | = | \mathcal{N} |$.

Proof. Let $S = \mathcal{N} \times \mathcal{N} = \{(k,j) \mid k,j \in \mathcal{N}\}$. 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, $\forall$ $k,j \in \mathcal{N}$, $f((k,j)) \geq 0$. We have to show that f is 1-1 and onto $\mathcal{N}$.
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 $j-j' \geq 0$). 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 $f((k,j)) \neq f((k',j')$.

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 $f((k,j)) \neq f((k',j')$. Thus f is 1-1.

To show that f is onto $\mathcal{N}$, we need to show that given any $m \geq 0$, there is a (k,j) such that $f((k,j)) = m$. Let x be the largest non-negative integer such that $x(x+1)/2 \leq m$. 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 $|\mathcal{N}| = |\mathcal{Q}|$

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

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

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

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

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

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

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

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

Assignment # 1

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
   $|S \times S \times S \times S \times S| \leq |\mathcal{P}(S)|$, 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. Consider the function *pair*: $\mathcal{N} \times \mathcal{N} \rightarrow \mathcal{N}$
   defined by $pair(x,y) = 2^x (2y + 1) - 1$
   Show that *pair* is a bijection (1-1 onto $\mathcal{N}$).
   Note that I already showed this is a surjection in the Sample, so your assignment is to show it is an injection (1-1), not just onto.

Due: Thursday, 8/28, at start of class (1:30PM)

Computability

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

Basic Definitions

The Preliminaries

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, Problems & Predicates

- 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".
- Predicate -- A mapping from some universe U into the Boolean set {true, false}. A predicate need not be defined for all values in U.

How They relate

- Let S be an arbitrary subset of some universe U. The predicate χ_S over U may be defined by:

  $\chi_S(x)$ = true if and only if $x \in S$

  χ_S is called the <u>characteristic function</u> of S.
- Let K be some arbitrary predicate defined over some universe U. The problem P_K associated with K is the problem to decide of an arbitrary member x of U, whether or not K(x) is true.
- Let P be an arbitrary decision problem and let U denote the set of questions in P (usually just the set over which a single variable part of the questions ranges). The set S_P associated with P is

  $\{ x \mid x \in U$ and x has answer "yes" in P $\}$

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

Existence of Undecidables

- A counting argument
  - The number of mappings from $\aleph$ to $\aleph$ is at least as great as the number of subsets of $\aleph$. But the number of subsets of $\aleph$ 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

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 ℵ, 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".

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 apply these techniques and results across Computer Science.

$\sqrt{p}$ is irrational

Prove, if p is a prime number, then $\sqrt{p}$ is irrational.
Hint: Look at Theorem 0.24 in Sipser.

Assume $\sqrt{p}$ is a rational number. Let q/r be the reduced fraction (no common prime factors) that equals $\sqrt{p}$.

$\sqrt{p} = q/r$: assumption
$r\sqrt{p} = q$: multiply both sides by r
$r^2 p = q^2$: square both sides

Since r and q have no common prime factors, then p must be a prime factor of q, so

$r^2 p = (kp)^2$: for some positive integer k
$r^2 = k^2 p$: divide both sides by p

Since r and q have no common prime factors, r and k have no common prime factor and so p must be a prime factor of r. But then q/r is not reduced as both q and r have the common prime factor p. This contradicts our original assumption that is $\sqrt{p}$ rational, so it is irrational.
QED

Assignment # 2

1. 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.
2. Present a language L over $\Sigma = \{a\}$ where $L^4 = L^5$ but $L \neq L^2$ and $L^2 \neq L^3$ and $L^3 \neq L^4$
   Note: $L^k = \{ x_1x_2\ldots x_k \mid x_1,x_2,\ldots,x_k \in L \}$

Due: Thursday, September 4, at start of class (1:30PM)

Complexity

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 problem as algorithmically solvable or not.
- Algorithm Design & Analysis tries to find the fastest possible data structures and algorithms to solve 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?

Regular Languages

Outline

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, concat, *, complement, reversal, intersection, set difference, substitution, homomorphism and inverse homomorphism, INIT, LAST, MID, EXTERIOR, quotient (with regular set, with arbitrary set).
- Algorithms for reachable states and states that can reach a point.
- Decision properties: Emptiness, finiteness, equivalence.

FSA and Sequential Circuits

- 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 defines state
  - Simple combinatorial circuits (and, or, not) that combine state and input to alter state
  - Simple combinatorial circuits (and, or, not) that use state to determine output

FSAs and Pattern Matching

- Will do some in class
- Think about FSA to recognize the string PAPAPAT appearing somewhere in a corpus of text, say with a substring PAPAPAPATRICK

Lexical Analysis

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

Game Behaviors

- Consider adding actions and weights on transitions
- Input to FSA enables some (possibility no) transitions from current state
- Each weight is a probability that a transition is fired if more than one is enabled
- Actions are initiated during transition
- Have an FSA per object, with communication occurring between FSAs and from environment, e.g., game controllers, trackers, etc.

Assignment # 3

1. Present a transition diagram for a NFA that recognizes the set of binary strings that starts with a 1 and, when interpreted as entering the DFA most to least significant digit, each represents a binary number that is divisible by either five or six. Thus, 101, 110, 1100, 1111 are in the language, but 111, 1011 and 11010 are not.
   OR
   Present a DFA that recognizes such binary strings that represent a number that is either 5 Mod 6 or 0 Mod 6.
2. a.) Present a transition diagram for an NFA for the language associated with the regular expression (1001 + 110 + 11)*. Your NFA must have no more than five 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.
3. 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.

Due: Thursday, September 11, at start of class (1:30PM)

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 Languages = Finite State 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

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

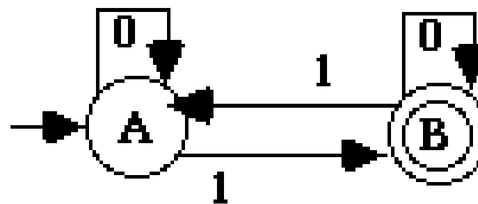
- We first show that QP* is contained in R. By definition, R = Q + RP.
- To see if QP* is a solution, we insert it as the value of R in Q + RP and see if the equation balances
- $R = Q + QP^*P = Q(\lambda+P^*P) = QP^*$
- Hence QP* is a solution, but not necessarily the only solution.

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

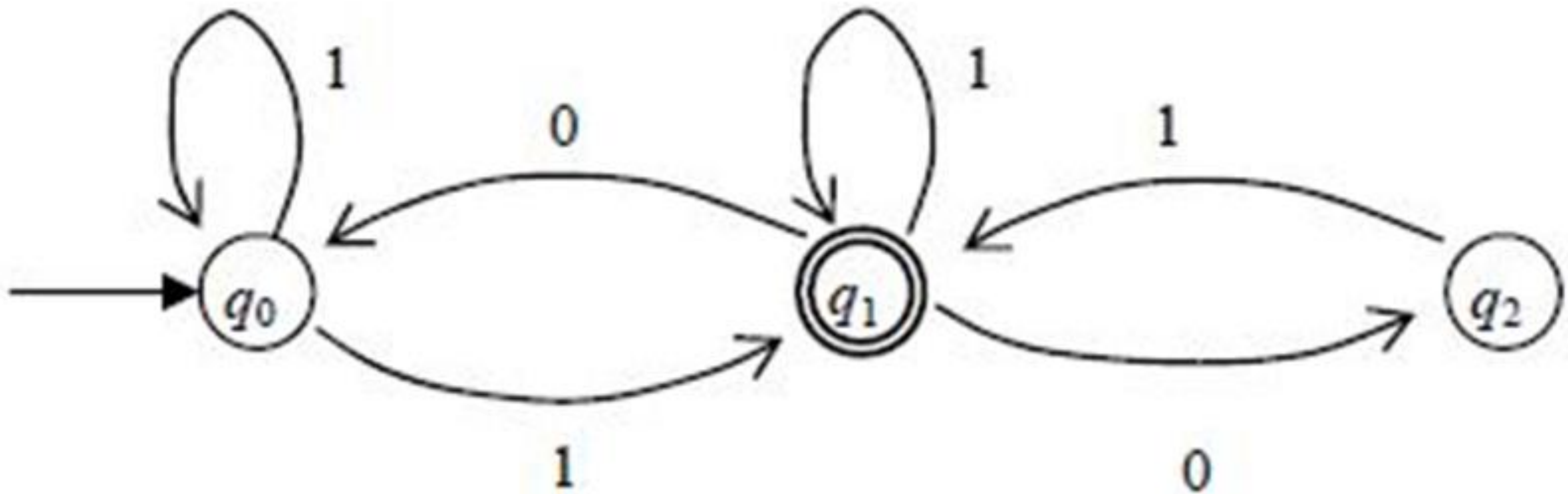
- We use the above to solve simultaneous regular equations. For example, we can associate regular expressions with finite state automata as follows



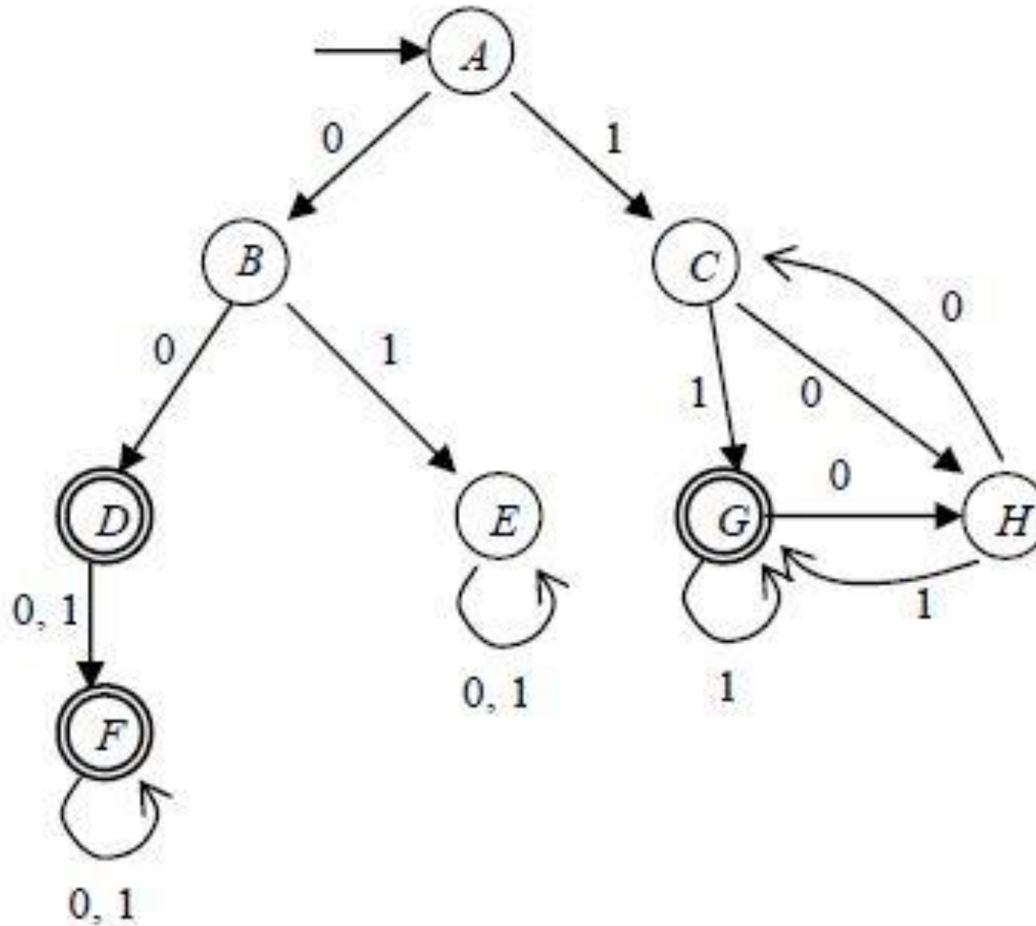
- Hence,
- A = B10* + 0*
- B = B10*1 + B0 + 0*1

- and therefore
- B = 0*1(10*1 + 0)*
- Note: This technique fails if there are lambda transitions.

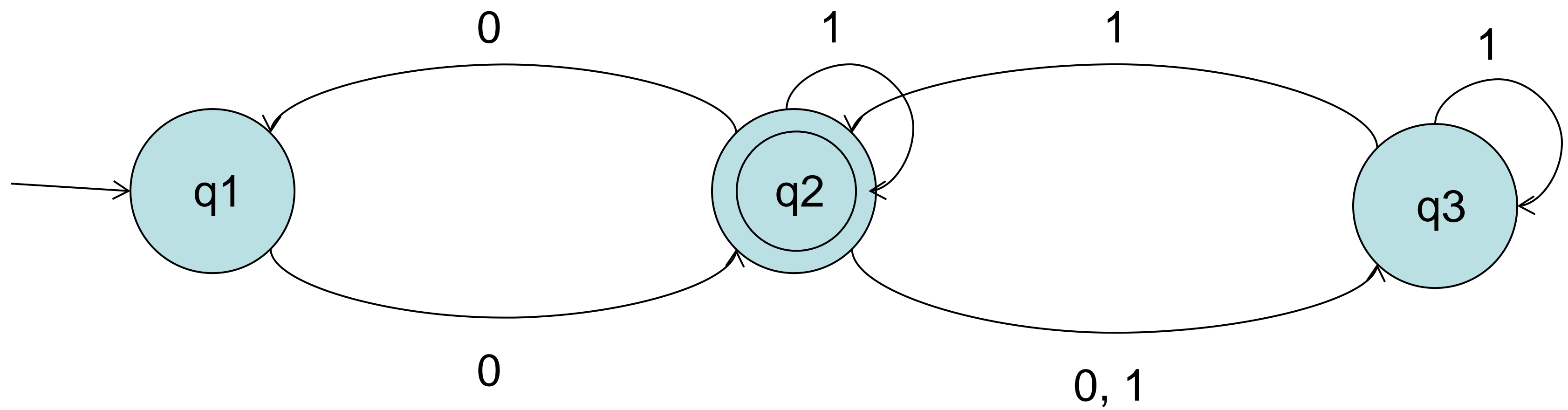
Convert from NFA to DFA

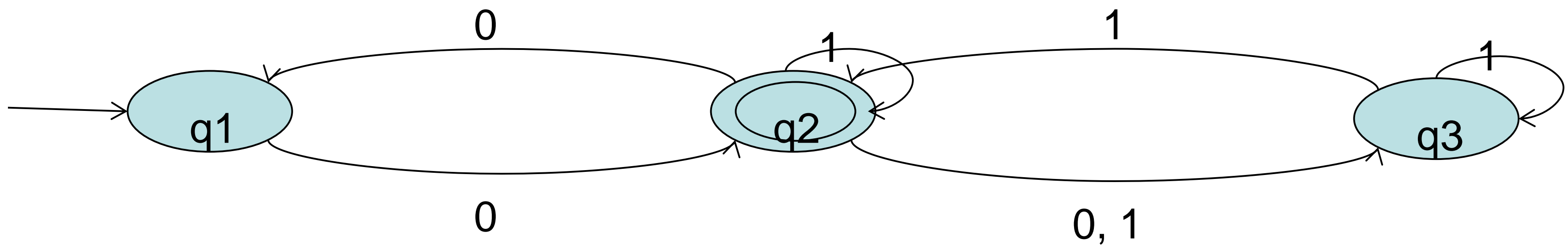


Minimize States



Convert to RE

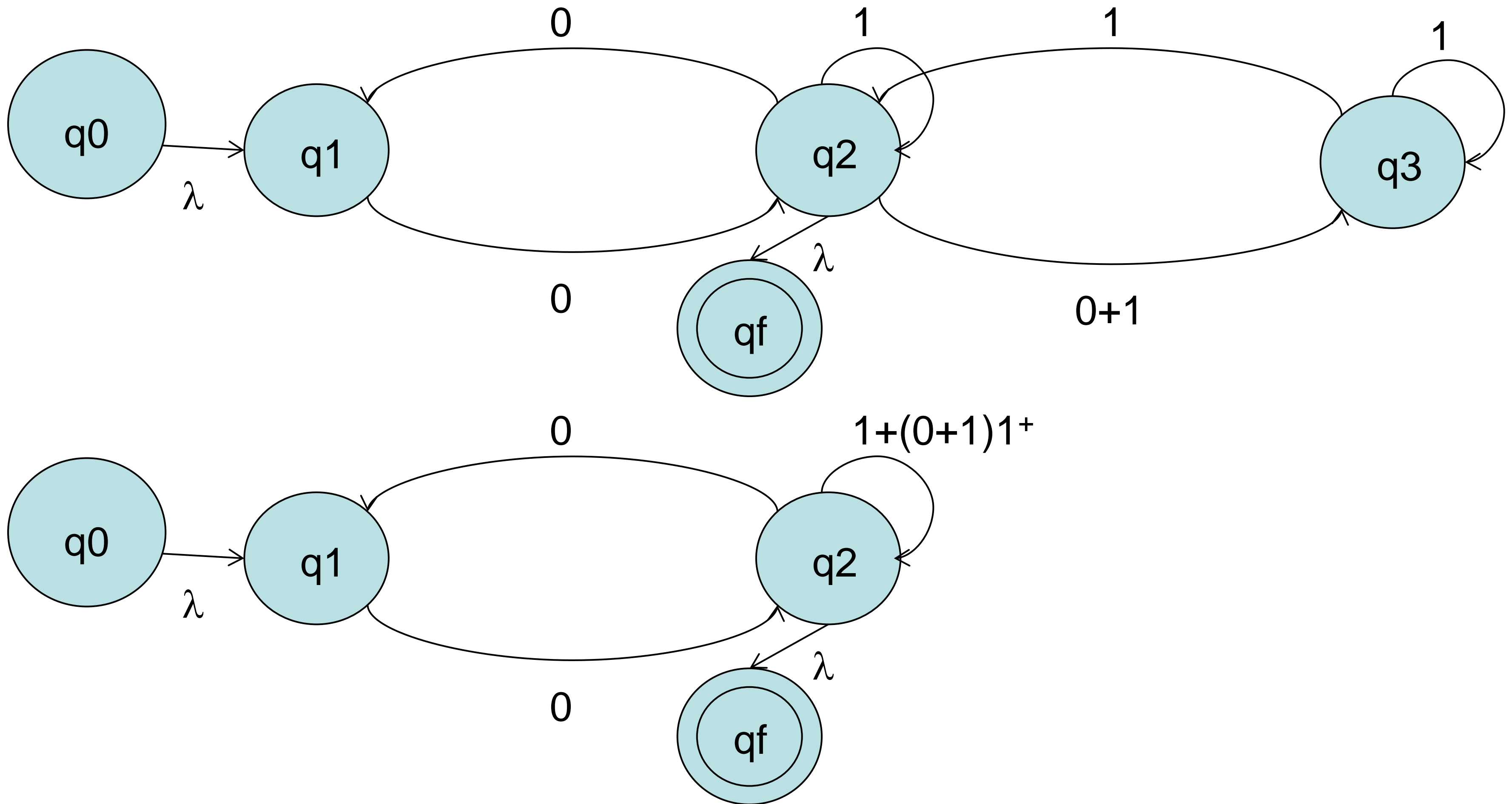




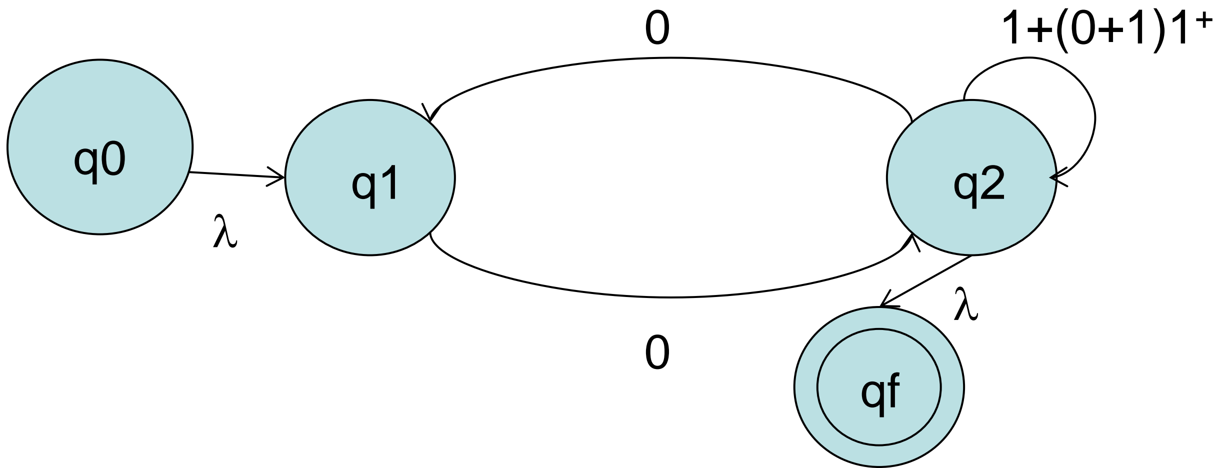
• $R_{11}^{0} = \lambda$	$R_{12}^{0} = 0$	$R_{13}^{0} = \phi$
• $R_{21}^{0} = 0$	$R_{22}^{0} = \lambda + 1$	$R_{23}^{0} = 0 + 1$
• $R_{31}^{0} = \phi$	$R_{32}^{0} = 1$	$R_{33}^{0} = \lambda + 1$
• $R_{11}^{1} = \lambda$	$R_{12}^{1} = 0$	$R_{13}^{1} = \phi$
• $R_{21}^{1} = 0$	$R_{22}^{1} = \lambda + 1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

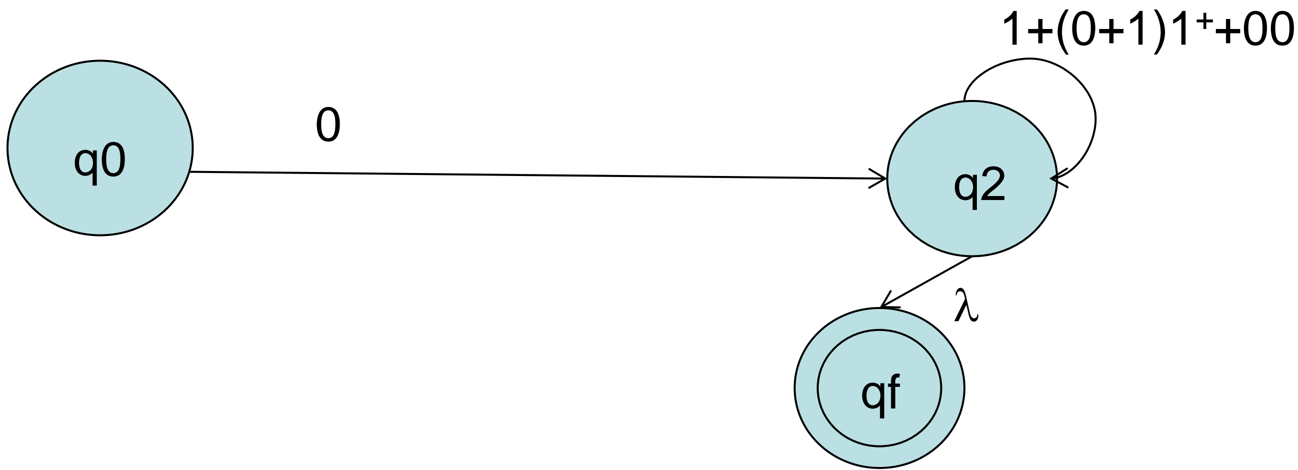
- $L = R_{12}^{3} =$
  $0(1+00)^* + 0(1+00)^*(0+1)\ (1+1(1+00)^*(0+1))^*\ 1(1+00)^*$

Use Ripping; Rip q3

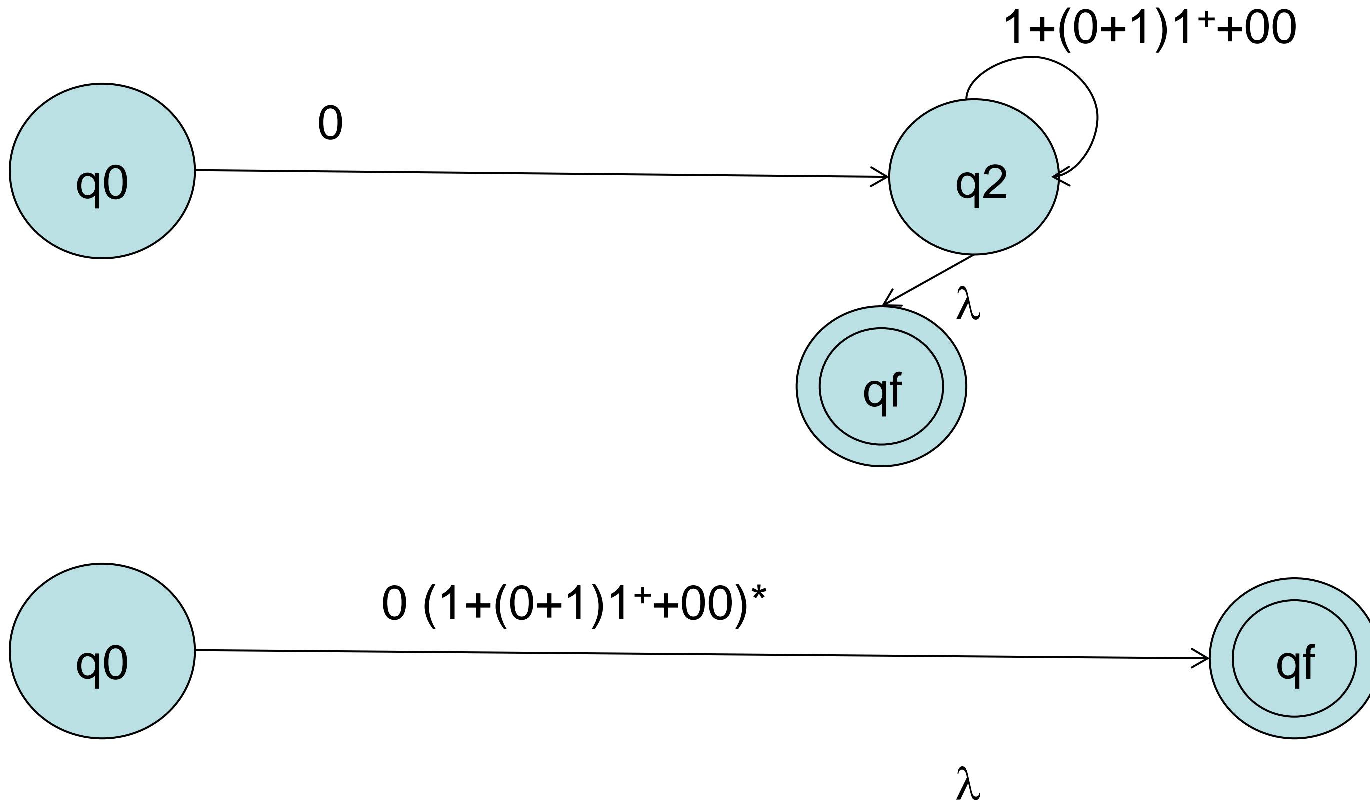


Use Ripping; Rip q1



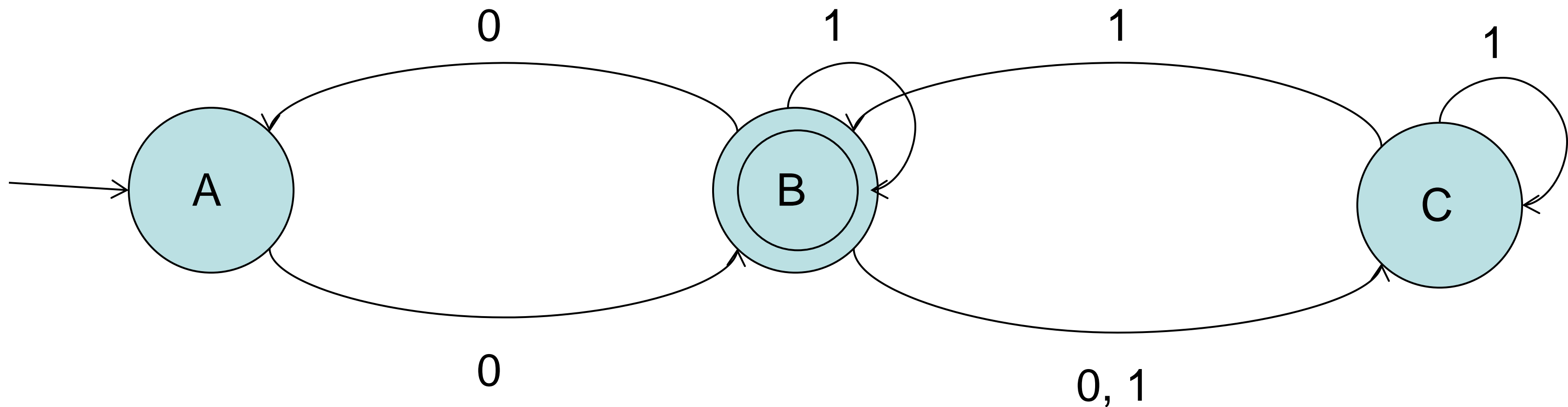


Use Ripping; Rip q2



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

Use Regular Equations



$A = \lambda + B0$
$B = A0 + C1 + B1$
$C = B(0+1) + C1;\ C = B(0+1)1^*$
$B = 0 + B00 + B(0+1)1^+ + B1$
$B = 0 + B\ (00+(0+1)\ 1^+ + 1);\ B = 0(00 +(0+1)1^+ + 1)^*$

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

Pumping Lemma Problems

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

NFAs

- Write NFAs for each of the following
  - $(111 + 000)^+$
  - $(0+1)^* \, 101 \, (0+1)^+$
  - $(1 \, (0+1)^* \, 0) + (0 \, (0+1)^* \, 1)$

Convert NFA to DFA

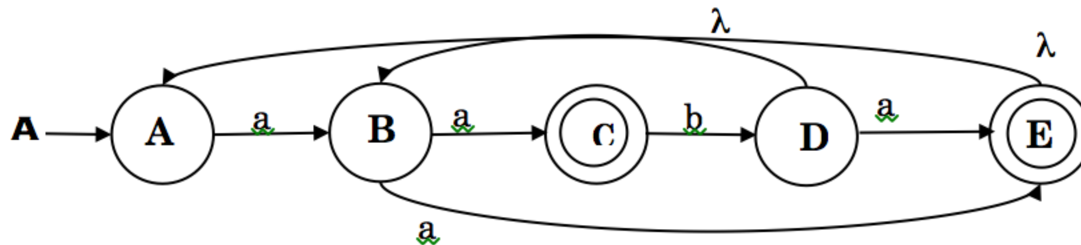
- Convert each NFA you just created to an equivalent DFA.

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 from two pages ago.

Assignment # 4

1. Convert the following NFA to an equivalent DFA.



2. Convert the DFA you developed in #1 to a regular expression, first by using either the GNFA (or state ripping) or Rij(k) approach, and then by using regular equations. You must show all steps in each part of this assignment.

Due: Thursday, September 18, at start of class (1:30PM).

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 More is Regular?

- Any language generated by a right linear grammar
- Any language generated by a left linear grammar
- Any language that is the union of some of the classes of a right invariant equivalence relation of finite index

What is NOT Regular?

- Well, I suppose 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

Pumping Lemma For Regular

- L is regular iff 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.

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 $w = v_1\ldots v_i\ldots v_j\ldots v_m$, where $m = |w|$, and $\delta(q_0,v_1\ldots v_i)=\delta(q_0,v_1\ldots v_j)$, j>I, and this state representing the first one repeated while A consumes w.
- Define $x = v_1\ldots v_i$, $y = v_{i+1}\ldots v_j$, and $z = v_{j+1}\ldots v_m$. Clearly w=xyz. Moreover, since $j > i$, $|y| > 0$, and since $j \leq N$, $|xy| \leq N$.
- Since A is deterministic, $\delta(q_0,xy)=\delta(q_0,xy^i)$, for all $i\geq 0$.
- Thus, if $w \in L$, $\delta(q_0,xyz) \in F$, and so $\delta(q_0,xy^iz) \in F$, for all $i\geq 0$.
- Consequently, if $w \in L$, $|w|\geq N$, then w can be written in the form xyz, where $|xy| \leq N$, $|y|>0$, and for all $i\geq 0$, $xy^iz \in L$.

Lemma's Adversarial Process

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

xwx is not Regular (PL)

- **L = { x w x | x,w∈{a,b}+} :**
- Assume that L is Regular.
- PL: Let N>0 be given by the Pumping Lemma.
- YOU: Let s be a string, s ∈ L, such that s = $a^N baa^N b$
- PL: Since s ∈ L and |s| ≥ N, s can be split into 3 pieces, s = xyz, such that |xy| ≤ N and |y| > 0 and ∀ i ≥ 0 xy^iz ∈ L
- YOU: Choose i = 2
- PL: xy^2z = xyyz ∈ L (could also use i = 0)
- Thus, $a^{N + |y|}baa^Nb$ would be in L, this is not so since N+|y| ≠ N and we cannot merge other a since must have |w|>0
- We have arrived at a contradiction.
- Therefore L is not Regular.

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.

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

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.

Finite State Transducers

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

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 two of these using the Pumping Lemma and at least two using Myhill-Nerode.

a. $\{ a^{Fib(k)} \mid k>0 \}$ This is set $\{a^1, a^1, a^2, a^3, a^5, a^8, a^{13}, a^{21}, \ldots \}$

b. $\{ a^i b^j c^k \mid i\geq 0, j\geq 0, k\geq 0, k = \min(i,j) \}$

c. $\{ a^i b^j c^k \mid i\geq 0, j\geq 0, k\geq 0, j = i * k \}$

d. $\{ a^i b^j c^k \mid i\geq 0, j\geq 0, k\geq 0, \text{if } i=1 \text{ then } j>k \}$

e. $\{ w \mid w \in \{a, b\}^* \text{ and } w = w^R \}$ this is the set of palindromes. It contains strings like aa, abba, abaaba

2. 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)

3. Write a regular (right linear) grammar that generates the set of strings denoted by the regular expression $((10)^+ (011 + 1)^+)^* (0+101)^*$

Due: Thursday, October 9, at start of class (1:30PM).

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 which can be expressed using BNF.

© UCF EECS

Grammars

- $G = (V, \Sigma, R, S)$ where
  - V: Finite set of non-terminal symbols
  - Σ: Finite set of terminal symbols
  - R: finite set of rules of form $\alpha \to \beta$,
    - 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 $\alpha \rightarrow \beta$
- $\Rightarrow^*$ is the reflexive, transitive closure of $\Rightarrow$
- $\Rightarrow+$ is the transitive closure of $\Rightarrow$
- $x \Rightarrow^* y$ iff $x = y$ or $x \Rightarrow^* z$ and $z \Rightarrow y$
- Or, $x \Rightarrow^* y$ iff $x = y$ or $x \Rightarrow z$ and $z \Rightarrow^* y$
- $L(G) = \{ w \mid S \Rightarrow^* w \}$ is the language generated by G.

Context Free Grammars

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

V is a finite set of symbols called the non-terminals or variables (sometimes denoted N). They are not part of the language generated by the grammar.

Σ is a finite set of symbols, disjoint from V, called the terminals. Strings in the language are made up entirely of terminal symbols.

S is a member of V and is called the start symbol.

R is a finite set of rules or productions. Each member of R is one 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)

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).

© UCF EECS

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>**

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**

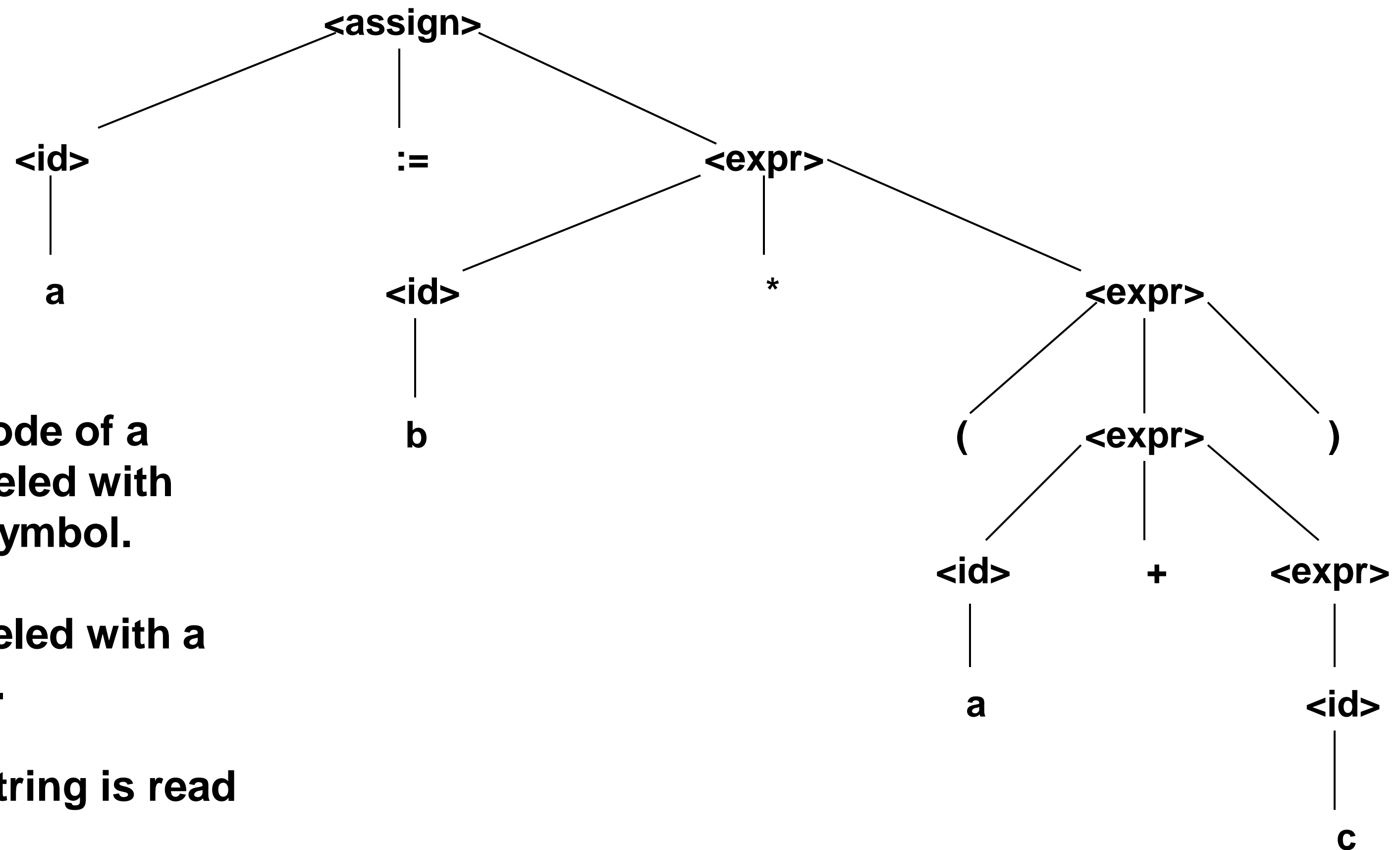
In a **leftmost derivation** only the leftmost non-terminal is replaced

© UCF EECS

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

© UCF EECS

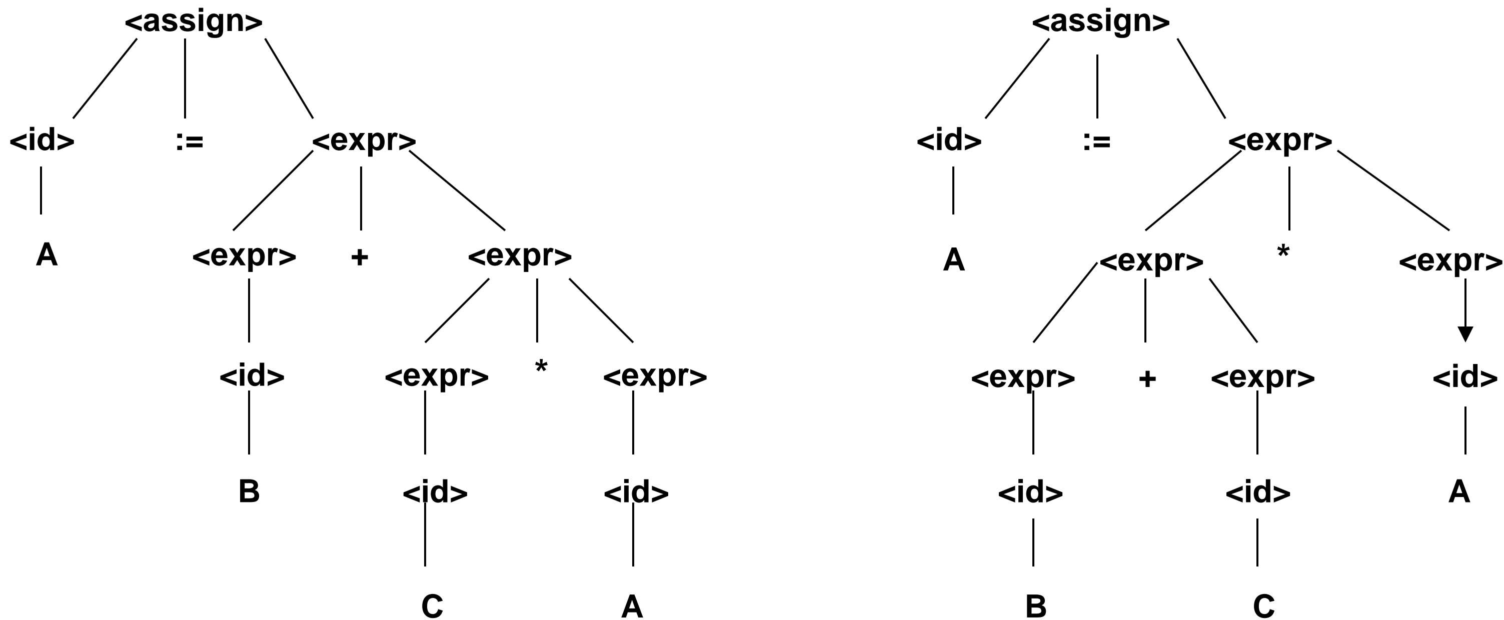
Ambiguity

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

© UCF EECS

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

© UCF EECS

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>)

© UCF EECS

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

© UCF EECS

Removing Left Recursion

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

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

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

© UCF EECS

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

© UCF EECS

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

© UCF EECS

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
- $\delta : Q \times \Sigma_e \times \Gamma_e \rightarrow 2^{Q \times \Gamma_e}$ is transition function
- Z_0 is initial symbol on stack
- $F \subseteq Q$ is final set of states

Notion of ID for PDA

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

Language Recognized by PDA

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

Top Down Parsing by PDA

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

Top Down Parsing by PDA

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid Int$

- $\delta(q,+,+)=\{(q,\lambda)\}$, $\delta(q,*,*)=\{(q,\lambda)\}$,
- $\delta(q,Int,Int)=\{(q,\lambda)\}$,
- $\delta(q,(,()=\{(q,\lambda)\}$, $\delta(q,),))=\{(q,\lambda)\}$
- $\delta(q,\lambda,E) = \{(q,E+T), (q,T)\}$
- $\delta(q,\lambda,T) = \{(q,T*F), (q,F)\}$
- $\delta(q,\lambda,F) = \{(q,(E)), (q,Int)\}$

Bottom Up Parsing by PDA

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

Bottom Up Parsing by PDA

E → E + T | T

T → T * F | F

F → (E) | Int

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

Challenge

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

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 $w \in \mathcal{F}(A_2)$ iff $t \in F_0$ and $s \in F_1$ iff $w \in \mathcal{F}(A_0)$ and $w \in \mathcal{F}(A_1)$

Substitution

- CFLs are closed under CFL substitution
  - Let $G=(V,\Sigma,R,S)$ be a CFG.
  - Let f be a substitution over Σ such that
    - $f(a) = L_a$ for $a \in \Sigma$
    - $G_a = (V_a,\Sigma_a,R_a,S_a)$ is a CFG that produces L_a.
    - No symbol appears in more than one of V or any V_a
  - Define $G_f = (V \cup_{a\in\Sigma}V_a, \cup_{a\in\Sigma}\Sigma_a, R' \cup_{a\in\Sigma}R_a, S)$
    - $R' = \{ A \rightarrow g(\alpha)$ where $A \rightarrow \alpha$ is in R $\}$
    - $g: (V\cup\Sigma)^* \rightarrow (V \cup_{a\in\Sigma}S_a)^*$
    - $g(\lambda) = \lambda$; $g(B) = B$, $B \in V$; $g(a) = S_a$, $a \in \Sigma$
    - $g(\alpha X) = g(\alpha)\ g(X)$, $|\alpha| > 1$, $X \in V\cup\Sigma$
  - 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 $P' = \{ A \to g(\alpha)$ where $A \to \alpha$ 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 $\in \mathscr{L}(G)$. But, then $w \in \mathscr{L}(G)$ iff $f(w) \in \mathscr{L}(G_f)$ and, thus, $f(\mathscr{L}(G)) = \mathscr{L}(G_f)$.

- Given that CFLs are closed under intersection, substitution, homomorphism and intersection with regular sets, we can recast previous proofs to show that CFLs are closed under
  - Prefix, Suffix, Substring, Quotient with Regular Sets

- Later we will show that CFLs are not closed under Quotient with CFLs.

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.

© UCF EECS

CKY (Bottom-Up Technique)

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

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

Let R_1 be the start symbol.

Let P[n,n,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 $a_1 \dots a_n$ is member of language

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

© UCF EECS

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			

© UCF EECS

2nd 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						

Converting a PDA to CFL

- 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
  - δ'(q_0', λ, \$) = {(q_0, 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 δ'(q_f, λ, \$) 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 CFL

- 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 $\delta(q,a,X) \subseteq \{(s,PUSH(Y))\}$
  $<q,X,p> \rightarrow a$
  whenever $\delta(q,a,X) \subseteq \{(p,POP)\}$
- Want $<q,X,p> \Rightarrow^* w$ when $[q,w,X] \vdash^* [p,\lambda,\lambda]$

Midterm#2 Topics

- Right-invariant equivalence relationships
  - Definition of RI Equiv
  - Myhill-Nerode Theorem
  - Existence of minimal state machine for any Regular Language
  - Application of Pumping Lemma for Regular languages

Midterm#2 Topics

- Right Linear Grammars
  - Definition and notion of derivation
  - Equivalence to finite automata
  - Closure properties
- Notion of instantaneous descriptions of machines and grammars
- Mealy and Moore Machines (automata with output); I will not ask any Moore questions
- Closure of regular under substitution
  - Use of substitution and intersection for other closures

Midterm#2 Topics

- Context free grammars
  - Writing grammars for specific languages
  - Leftmost and rightmost derivations, Parse trees, Ambiguity
  - Closure (union, concatenation, substitution)
  - Non-closure (intersection and complement)
  - 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
  - Top-down vs bottom up parsing
- Closure
  - Intersection with regular
  - Quotient with regular, Prefix, Suffix, Substring
- Non-Closure
  - Intersection, complement, min, max

Computability

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

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 ℵ, 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".

Immediate Implications

- P solved implies P solvable implies P semi-decidable (re).
- 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 S_P and $(U — S_P)$ are re (semi-decidable).

- We will prove these later.

Hilbert's Tenth

Diophantine Equations are Unsolvable

One Variable Diophantine Equations are Solvable

Hilbert's 10th is Semi-Decidable

- Consider over one variable: $P(x) = 0$
- Can semi-decide by plugging in
  0, 1, -1, 2, -2, 3, -3, …
- 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

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

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

Basic Description

- We will use a simplified form that is a variant of Post's and Turing's models.
- Here, each machine is represented by a finite set of states 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 print 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)

…?11…10… ⇒ …?11…10…

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

…011…1?… ⇒ …011…1?…

Commentary on Machines

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

Computing with TMs

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

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

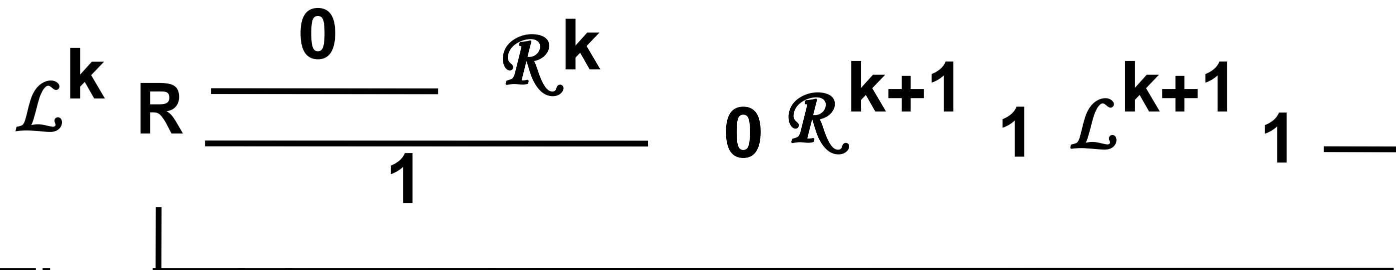
and ends with

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

where y = F(x1, … , xn).

Addition by TM

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



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:

  INC_r[i] –

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

  DEC_r[p, z] –

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

  else jump to **z**

- Note, we do not use subscripts if obvious.

© UCF EECS

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 $3^{x1}5^{x2}\ldots p_n^{xn}$. The result then appears as the power of the prime **2**.

© UCF EECS

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

$\mathbf{bx \rightarrow ax}$

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

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

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

© UCF EECS

Limited Subtraction by FRS

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

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

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

$\mathbf{5x \rightarrow x}$

© UCF EECS

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.

© UCF EECS

Why Deterministic?

To see why determinism makes a difference, consider

$3 \cdot 5x \rightarrow x$

$3x \rightarrow 2x$

$5x \rightarrow x$

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

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

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

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

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

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

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

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

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

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

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

…

This computes **2^z** where **$0 \le z \le x_1$**. 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**.

© UCF EECS

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.

$3*3\ x \rightarrow x$

$3\ x \rightarrow 2\ x$

Sample FRS

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

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

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

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

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

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

$11\ x \rightarrow 5\ x$

$5\ x \rightarrow x$

$7\ x \rightarrow x$

Systems Related to FRS

- Petri Nets:
  - Unordered
  - Ordered
  - Negated Arcs
- Vector Addition Systems:
  - Unordered
  - Ordered
- Factors with Residues:
  - $a\,x + c \;\rightarrow\; b\,x + d$

Petri Net Operation

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

Petri Net Computation

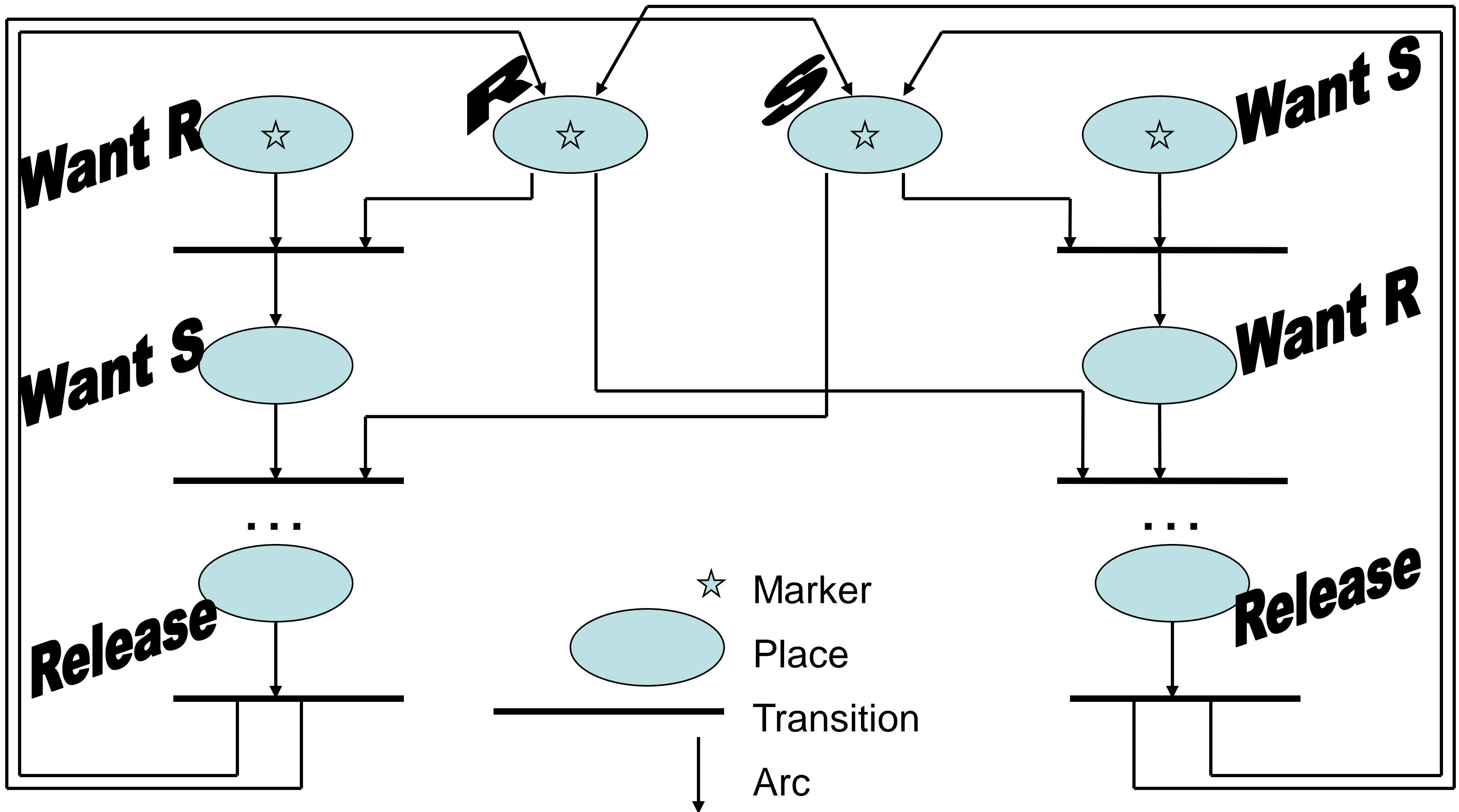
- A Petri Net starts with some finite number of markers distributed throughout its **n** nodes.
- The state of the net is a vector of n natural numbers, with the **i**-th component's number indicating the contents of the **i**-th node. E.g., **<0,1,4,0,6>** could be the state of a Petri Net with **5**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

 © UCF EECS

Variants of Petri Nets

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

Petri Net Example



Vector Addition

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

Vectors as Resource Models

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

© UCF EECS

Factors with Residues

- Rules are of form
  - $a_i\, x + c_i \;\rightarrow\; b_i\, x + d_i$
  - There are **n** such rules
  - Can apply if number is such that you get a residue (remainder) c_i when you divide by a_i
  - Take quotient **x** and produce a new number $b_i\, x + d_i$
  - Can apply any applicable one (no order)
- These systems are equivalent to Register Machines.

Undecidability

We Can't Do It All

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." 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)***↓.

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)***↑.

Of course, if ***f(x)***↓ then ***f*** defines a value for ***x***. In fact we also say that ***f(x)*** is defined if ***f(x)***↓ and undefined if ***f(x)***↑.

Finally, we define the domain of ***f*** as **{*x* | *f(x)*↓}**.
The range of ***f*** is **{*y* |** there exists an x, ***f(x)***↓ and ***f(x)* = *y*** }.

Halting Problem

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

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

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

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

with inverses

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

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

The Contradiction

Now if Halt exist, then so does Disagree, where

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

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

Disagree(d) is defined ⇔ Halt(d,d) = 0
⇔ [d](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 is not solvable.

Halting is recognizable

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

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

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

Why not just algorithms?

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

Here's the logic. Any programming language needs to have an associated grammar that can be used to generate all legitimate programs. By ordering the rules of the grammar in a way that generates programs in some lexical or syntactic order, we have a means to recursively enumerate the set of all programs. Thus, the set of procedures (programs) is re. using this fact, we will employ the notation that φ_x is the **x**-th procedure and φ_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,

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

Non-re Problems

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

  $F(x) = F_x$

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

The Contradiction

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

Enumeration Theorem

- Define

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

- Theorem: A set **B** is re iff there exists an **n** such that $\mathbf{B = W_n}$.
  Proof: Follows from definition of φ**(n,x)**.
- This gives us a way to enumerate the recursively enumerable (semi-decidable) sets.

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 for 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 Petri Nets, (Ordered) Factor Replacement Systems

Non-determinism

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

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> | φ_f (x)$\downarrow$ } is unsolvable (undecidable, non-recursive)
- TOTAL = { f | $\forall$x φ_f (x)$\downarrow$ } = { f | W_f =***N*** } is not even recursively enumerable (re, semidecidable)

- Show ZERO = { f | $\forall$x φ_f (x) = 0 } is unsolvable.
  <f,x> $\in$ HALT iff g(y) = φ_f (x) - φ_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 | $\forall$x φ_f (x) = 0 } is non-re.
  <f> $\in$ TOTAL iff h(x) = φ_f (x) - φ_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.

Assignment # 7

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 { f | ∃x f(x) = x } is undecidable**
2. **Show that { f | ∃x f(x) = x } reduces to Halt. (1 plus 2 show they are equally hard)**
3. **Use reduction from Halt to show that one cannot decide { f | ∀x f(x+1)=2*f(x)+1} Note that f(0) can be any value.**
4. **Use Reduction from Total to show that { f | ∀x f(x+1)=2*f(x)+1} is not even re**
5. **Show { f | ∀x f(x+1)=2*f(x)+1} reduces to Total. (4 plus 5 show they are equally hard)**

Due: **November 18, at start of class (1:30PM).**

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: $\varnothing$, ***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 $\in$ dom(φ_a) iff φ_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

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

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

$L_P = \{ x \mid dom [x] = dom\ \varphi_x$ 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.

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 $[r] = \varphi_r$ be a recursive function whose domain is L (r is the index of a semi-decision procedure for L). Suppose P were decidable. We will use this decision procedure and the existence of r to decide K_0. First we define a function $F_{r,x,y}$ for r and each function $[x] = \varphi_x$ and input y as follows.

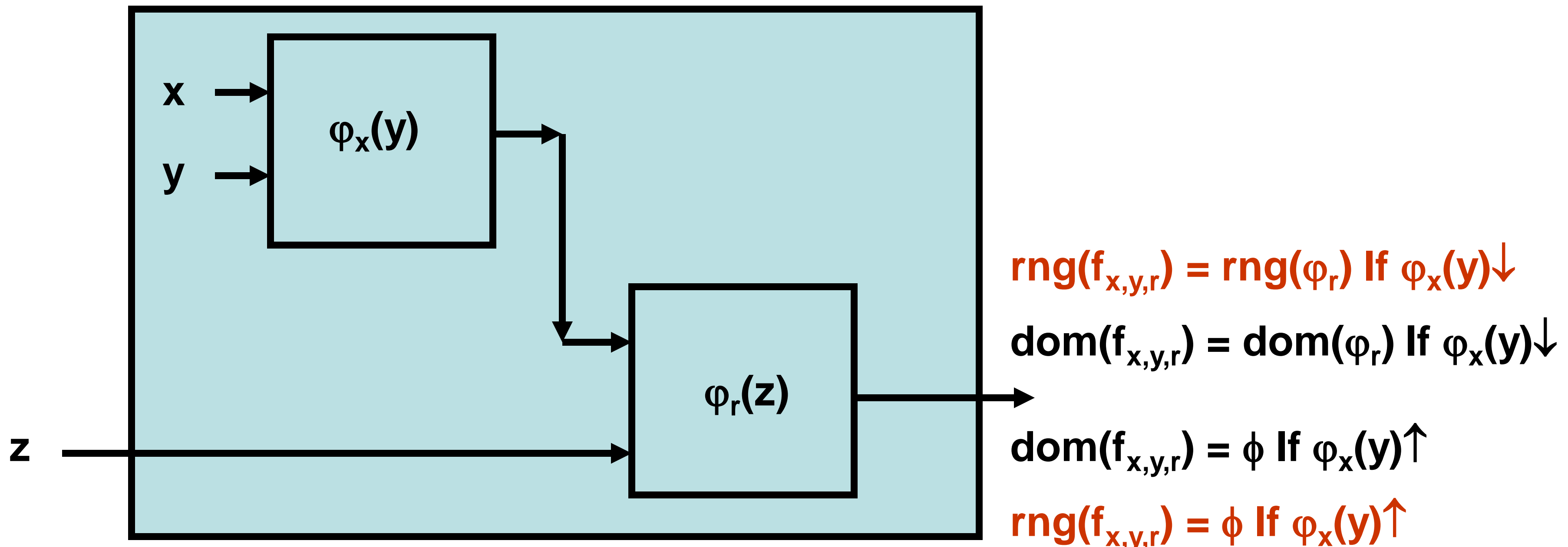
$F_{r,x,y}(z) = \varphi_x(y) + \varphi_r(z)$

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

Rice's Picture Proof

Let $\mathscr{P}$ be an arbitrary, non-trivial, I/O property of effective procedures. Assume wlog that the functions with empty domains are not in $\mathscr{P}$.

Given **x**, **y**, **r**, where **r** is in the set $\mathbf{S}_{\mathscr{P}}$ **= {f |** φ_f has property $\mathscr{P}$}, define the function $\mathbf{f_{x,y,r}(z) = \varphi_x(y)} - \varphi_x(y) + \varphi_r(z)$. The following illustrates $\mathbf{f_{x,y,r}}$.
Here, $\mathbf{dom(f_{x,y,r}) = dom(\varphi_r)}$ $(f_{x,y,r}(z) = \varphi_r(z))$ **if** $\varphi_x(y)\downarrow$; $= \phi$ **if** $\varphi_x(y)\uparrow$.
Thus, $\varphi_x(y)\downarrow$ iff $\mathbf{f_{x,y,r}}$ has property $\mathscr{P}$, and so $\mathbf{K_0 \leq_1 S_{\mathscr{P}}}$.



Corollaries to Rice’s

Corollary: The following properties of re sets are undecidable

a) $L = \varnothing$

b) L is finite

c) L is a regular set

d) L is a context-free set

Recursively Enumerable

Properties of re Sets

Definition of re

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

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

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

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

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

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

© UCF EECS

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 $\mathbf{G_S}$. Assume $\mathbf{G_S}$ is the $\mathbf{g_S}$ 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 $\boldsymbol{a} \in \mathbf{S}$. Define the enumerating algorithm $\mathbf{F_S}$ by

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

Note: $\mathbf{F_S}$ is <u>primitive recursive</u> 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

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

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 ψ_S we defined earlier to semi-decide **S**, given its enumerating function, can be easily adapted to have this property.

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

© UCF EECS

Recursive Implies re

Theorem: Recursive implies re.

Proof: **S** is recursive implies there is an algorithm **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

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

© UCF EECS

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** $\neq\emptyset$ then the following are equivalent:

1. **S** is re
2. **S** is the range of a primitive rec. function
3. **S** is the range of a recursive function
4. **S** is the range of a partial rec. function
5. **S** is the domain of a partial rec. function

© UCF EECS

Quantification#1

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

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

© UCF EECS

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,

  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.

© UCF EECS

Practice Assignment # 8

1. Use Rice's Theorem to show that $\{ f \mid \exists x\ f(x) = 0 \}$ is undecidable
2. Use Rice's Theorem to show that $\{f \mid \forall x\ f(x+1)=f(x)+1\}$ is undecidable
3. Use quantification of an algorithmic predicate to estimate the complexity (decidable, re, co-re, non-re) of each of the following, (a)-(d):
   a) $\{ f \mid$ for all input x, $f(x) = f(0)$, that is f is a constant function $\}$
   b) $\{ f \mid$ for two unique input values, x,y, $f(x) = f(y) \}$
   c) $\{ \langle f,x \rangle \mid f(x)$ takes at least 10 time steps before converging $\}$
   d) $\{ \langle f,x \rangle \mid f(x)\uparrow \}$
4. Let sets A and B each be re non-recursive (undecidable). Consider $C = A \cap B$. For (a)-(c), either show sets A and B with the specified property or demonstrate that this property cannot hold.
   a) Can C be recursive?
   b) Can C be re non-recursive (undecidable)?
   c) Can C be non-re?

Sample Question#1

1. Given that the predicate **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)**.

© UCF EECS

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**.

© UCF EECS

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 $\in$ **S and y** $\in$ **T }.**

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

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

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

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

© UCF EECS

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 listing re sets as Turing machines are at enumerating re sets (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:
$n = 3$, $\Sigma = \{ a , b \}$, (a b a , b b , a), (b a b , b , b a a).
Solution 2 , 3, 1 , 2
b b a a b a b b = b b a a b a b b
- In general, PCP is undecidable (no proof will be given)

PCP is undecidable

- We will not prove this here, but 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

- Problem to determine if an arbitrary CFG is ambiguous

  $S \to A \mid B$

  $A \to x_i\ A\ [i] \mid x_i\ [i] \qquad 1 \le i \le n$

  $B \to y_i\ B\ [i] \mid y_i\ [i] \qquad 1 \le i \le n$

  $A \Rightarrow^* x_{i_1} \dots x_{i_k} [i_k] \dots [i_1] \qquad k > 0$

  $B \Rightarrow^* y_{i_1} \dots y_{i_k} [i_k] \dots [i_1] \qquad k > 0$

- Ambiguous if and only if there is a solution to this PCP instance.

Intersection of CFLs

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

Non-emptiness of CSL

$S \rightarrow x_i\ S\ y_i^R \mid x_i\ T\ y_i^R \quad 1 \le i \le n$

$a\ T\ a \rightarrow *\ T\ *$

$*\ a \rightarrow a\ *$

$a\ * \rightarrow *\ a$

$T \rightarrow *$

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

Traces (Valid Computations)

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

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

  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. For example we might use reversals on the odd strings so the relation between each pair is context free.
- Many texts show that a context free grammar can be devised which approximates traces by either getting the even-odd pairs right, or the odd-even pairs right. The goal is to then to intersect the two languages, so the result is a trace. This then allows us to create CFLs 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.

One step traces

- The set of one step traces of a machine, M, is

  $\{ X_0 \# X_1 \}$

  where $X_0 \Rightarrow X_1$
- If we are considering Turing Machines, we use $\{ X_0 \# X_1^R \}$

  where $X_0 \Rightarrow X_1$ and X_1^R is the reversal of X_1
- By using the reversal we make the language no harder than $W \# W^R$, which is a CFL.

Turing Machine Traces

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

What’s Context Free?

- Given a Turing Machine M
  - The set of invalid traces of M is Context Free
  - The set of valid traces is Context Sensitive
  - The set of valid terminating traces is Context Sensitive
  - The complement of the valid traces is Context Free
  - The complement of the valid terminating traces is Context Free

Partially correct traces

L1 = L(G1) = { $\#Y_0 \# Y_1 \# Y_2 \# Y_3 \# \ldots \# Y_{2j} \# Y_{2j+1} \#$ }
where $Y_{2i} \Rightarrow Y_{2i+1}$, $0 \le i \le j$.
This checks the even/odd steps of an even length computation.
But, L2 = L(G2) = {$\#X_0\#X_1\#X_2\#X_3\#X_4 \#\ldots\# X_{2k-1}\#X_{2k}\#Z_0\#$}
where $X_{2i-1} \Rightarrow X_{2i}$, $1 \le i \le k$.
This checks the odd/steps of an even length computation.

L = L1 $\cap$ L2 describes correct traces (checked even/odd and odd/even). If Z_0 is chosen to be a terminal configuration, then these are terminating traces. If we pick a fixed X_0, then X_0 is a halting configuration iff L is non-empty. This is an independent proof of the undecidability of the non-empty intersection problem for CFGs and the non-emptiness problem for CSGs.

What’s Undecidable?

- We cannot decide if the set of valid terminating traces of an arbitrary machine M is non-empty.
- We cannot decide if the complement of the set of valid terminating traces of an arbitrary machine M is everything. In fact, this is not even semi-decidable.

$L = \Sigma^*$?

- If L is regular, then $L = \Sigma^*$? is decidable
  - Easy – Reduce to minimal deterministic FSA, $\mathcal{A}_L$ accepting L. $L = \Sigma^*$ iff $\mathcal{A}_L$ is a one-state machine, whose only state is accepting
- If L is context free, then $L = \Sigma^*$? is undecidable
  - Just produce the complement of a Turing Machine's valid terminating traces

Quotients of CFLs

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.
But, 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.

Traces and Type 0

- Assume we are given some machine M, with Turing table T (using Post notation). We assume a tape alphabet of Σ that includes a blank symbol B.
- Consider a starting configuration C_0. Our rules will be

S	→	**# C_0 #**	**where C0 = Yq_0aX 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 $\lvert\alpha\rvert \le \lvert\beta\rvert$**
α	$\rightarrow$	βD^k	**where $\alpha \rightarrow \beta \in$ P and $\lvert\alpha\rvert - \lvert\beta\rvert = k > 0$**

- 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.

Summary of Grammar Results

Decidability

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

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?
- Does there exist a finite n, $L^n=L^{N+1}$?
- Is L_1/L_2 finite, L_1 and L_2 CFLs?
- Membership in L_1/L_2, L_1 and L_2 CFLs?

Computational Complexity

Limited to Concepts of P and NP

COT6410 covers much more

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.

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.

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.

Satisfiability

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

(CNF – Conjunctive Normal Form)

$C = \{c_1, c_2, \ldots, c_m\}$, conjunction(anding) of "OR clauses"

Example clause:

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

Satisfiability

Can we assign Boolean values to the variables in U so that every clause is TRUE?

There is no known polynomial algorithm!!

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.

Simulating ND TM

- Given a TM, 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**

Cook's Theorem

- $\varphi_{M,w} = \phi\text{cell} \wedge \phi\text{start} \wedge \phi\text{move} \wedge \phi\text{accept}$
- See the following for a detailed description and discussion of the four terms that make up this formula.
- http://www.cs.tau.ac.il/~safra/Complexity/Cook.ppt

NP–Complete

Since SAT is itself in NP, that means SAT is a hardest problem in NP (there can be more than one.).

As with RE problems, a hardest problem in a class is called the "completion" of that class.

Therefore, SAT is NP–Complete.

NP–Complete

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

These included such problems as:

- **3-SAT**
- **3DM**
- **Vertex Cover,**
- **Independent Set,**
- **Knapsack,**
- **Multiprocessor Scheduling, and**
- **Partition.**

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 and Partition

Theorem. SAT $\square_P$ 3SAT

Theorem. 3SAT $\square_P$ SubsetSum

Theorem. SubsetSum $\square_P$ Partition

Theorem. Partition $\square_P$ SubsetSum

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

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

Example SubsetSum

Assuming a **3SAT expression (a + ~b + c) (~a + b + ~c), the following shows the reduction from 3SAT to Subset-Sum.**

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

Partition

- 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

2 Processor scheduling

The problem of optimally scheduling n tasks $T_1, T_2, \ldots, T_n$ 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

© UCF EECS

NP–Complete

Today, there are 1,000's of problems that have been proven to be NP–Complete. (See Garey and Johnson, *Computers and Intractability: A Guide to the Theory of NP–Completeness*, for a list of over 300 as of the early 1980's).

P = NP?

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

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

P = NP?

Why should P equal NP?

There seems to be a huge "gap" between the known problems in P and Exponential. That is, almost all known polynomial problems are no worse than n^3 or n^4.

Where are the $O(n^{50})$ problems?? $O(n^{100})$? Maybe they are the ones in NP–Complete?

It's awfully hard to envision a problem that would require n^{100}, but surely they exist?

Some of the problems in NP–C just look like we should be able to find a polynomial solution (looks can be deceiving, though).

$P \neq NP$?

Why should P not equal NP?

- P = NP would mean, for any problem in NP, that it is just as easy to solve an instance 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.

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).

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.

Final Exam Topics 1

- Regular languages
  - Finite State Automata: Deterministic and Non-Deterministic
  - Right Linear Grammars
  - Regular Expressions
  - Regular Equations
  - Right invariant equivalence relations of finite index
  - Equivalence of above six models
  - Closures: negation, union, exclusive or, concatenation, star, intersection, substitution, quotient, prefix, suffix, substring
  - MyHill-Nerode and minimum state DFA
  - Minimizing DFAs
  - Classic non-regular languages {0^n 1^n | n >= 0}
  - Pumping Lemma for Regular Languages
  - Notion of instantaneous descriptions of machines and grammars
  - Mealy and Moore Machines (automata with output)

Final Exam Topics 2

- Context free languages
  - Context free grammars
    - Leftmost and rightmost derivations
    - Parse trees
    - Ambiguity
  - Closure: union, concatenation, star, substitution, intersection with regular, quotient, prefix, suffix, substring
  - Non-closure: intersection, complement, quotient)
  - Pumping Lemma for CFLs
  - Chomsky Normal Form
    - Remove non-generating non-terminals (and rules)
    - Remove unreachable non-terminals (and rules)
    - Remove lambda rules
    - Remove chain rules
    - Make right-hand sides match CNF constraints
  - CKY algorithm
  - Push-down automata
  - Various notions of acceptance and their equivalence
  - Deterministic vs non-deterministic
  - Equivalence to CFLs (Proof that every PDA recognized language is a CFL is off the table)
  - Top-down vs bottom up parsing

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
  - 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
  - Rice's Theorem: All non-trivial I/O properties of functions are undecidable

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 not done and is not part of this course)
    - Application to ambiguity and non-emptiness of intersections of CFLs and to non-emptiness of CSL
  - 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
  - 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

Final Exam Topics 6

- Complexity Theory
  - Verifiers versus solvers: P versus NP
  - Definitions of NP: verify in det poly time vs solve in non-det poly 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, Partition
  - Relation to multiprocessor scheduling