Formal Languages and Automata Theory

COT 5310 – Fall 2007
Notes

Who, What, Where and When

- **Instructor:** Charles Hughes;
  Harris Engineering 439C; 823-2762;
  ceh@cs.ucf.edu
- **Web Page:** http://www.cs.ucf.edu/courses/cot5310/
- **Meetings:** MW 7:30PM-8:45PM, HEC-103;
  29 periods, each 75 minutes long.

  Office Hours: MW 5:00PM-6:15PM
- **GTA:** Greg Tener

  Office Hours: TR 6:30PM-7:30PM

Text Material

- **This and other material linked from web site. I will occasionally use Dr. Tiplea's notes from Fall 2005, as well as mine.**
- **References:**
  - **Davis, Sigal and Weyuker, *Computability, Complexity and Languages 2nd Ed.*, Academic Press (Morgan Kaufmann), 1994.**
  - **Hopcroft, Motwani and Ullman, *Introduction to Automata Theory, Languages and Computation 2nd Ed.*, Addison-Wesley, 2001.**

Expectations

- **Prerequisites:** COP 4020 (Covers parsing and some semantic models); COT 4210 (covers regular and context free languages)
- **Assignments:** Seven (7) or so. At least one (the review on prerequisite formal languages and automata) will be extensive.
- **Exams:** Two (2) midterms and a final.
- **Material:** I will draw heavily from Davis, Chapters 2-4, parts of 5, 6-8 and 11. Some material will also come from Hopcroft. Class notes and in-class discussions are, however, comprehensive and cover models and undecidable problems that are not addressed in either of these texts.

Goals of Course

- **Provide characterizations (computational models) of the class of effective procedures / algorithms.**
- **Study the boundaries between complete (or so it seems) 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.**
- **Apply results to various other areas of CS.**

© UCF (Charles E. Hughes)

Expected Outcomes

- **By the time this course ends, I expect you to have a solid understanding of models of computation, the limits that are imposed by the very nature of computation, and the ubiquity of unsolvable problems throughout CS.**
- **I also hope that you come away with stronger formal proof skills and a better appreciation of the importance of discrete mathematics to all aspects of CS.**
- **I do not expect to turn you into recursive function theorists. That's a long journey, of which this course represents only the first few steps.**

Being Prepared

- Prerequisites are COT 4210 and COP 4020.
- While I understand that some of you may not have the same material in your background as covered here, I do expect you to become familiar with the material in these courses.
- I will not spend time on the basics of formal languages, automata theory, or parsing.
- I will, however, approach the course material starting with computation theory, rather than the applications of theory to formal languages. You will have about six weeks to get on top of these topics before they become critical to your understanding of COT 5310.
- Use this time wisely to review or learn the prerequisite topics.

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 role.
- 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 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 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.

© UCF (Charles E. Hughes)

Grading

- **Grading of Assignments**
  - My GTA and I will grade harder than our actual expectations run. Consequently, a grade of 90% or above will translate into a perfect grade. In general, I will award everyone 10% over the grade they are assigned on the returned papers.
- **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 – Mon., October 1**
- **Withdraw Deadline – Fri., October 12**
- **Exam#2 – Wed., November 7**
- **Final – Wed., Dec. 5, 7:00PM – 9:50PM**
- **Holidays**
  - Labor Day – September 3
  - Veterans Day – November 12

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%, B+ >= 87%, B >= 80%, C+ >= 77%, C >= 70%, D >= 50%, F < 50%**

What You Should Know

- Proof Techniques
- Regular Sets
- Context Free Languages

Regular Sets # 1

- **Regular expressions: Definition and associated languages.**
- **Finite state automata. Associating FSAs with REs.**
- **Associating REs with FSAs. Proof using R_{ijk} sets.**
- **Moore and Mealy models: Automata with output. Basic equivalence.**
- **Non-determinism: Its use. Conversion to deterministic FSAs. Formal proof of equivalence.**
- **Lambda moves: Lambda closure of a state. Equivalence to non-determinism.**

Regular Sets # 2

- **Regular equations: REs 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.**

© UCF (Charles E. Hughes)

Regular Sets # 3

- **Pumping Lemmas: Proof and applications.**
- **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.**

Context Free # 1

- Leftmost versus rightmost derivations.
- Parse trees, A-trees. Definition of a parse tree and proof that A =>* X iff there exists an A-tree with X as its yield.
- Ambiguity and leftmost (rightmost) derivations.
- Pushdown automata; various notions of acceptance and their equivalences
- Push down languages and their equivalence to CFLs.
- Parsing Techniques: LL (top down) and LR (bottom up) parsers

Context Free # 2

- **Reduced grammars.**
- **Keep non-terminal A iff A =>* w for some terminal string w.**
- **Keep symbol X iff S =>* WXY for some strings W and Y.**
- **Lamba rule removal.**
- **Chain (unit) rule removal.**
- **Chomsky Normal Form.**
- **Left recursion removal.**
- **Greibach Normal Form.**

© UCF (Charles E. Hughes)

Context Free # 3

- Pumping Lemmas for CFLs.
- Closure of CFLs: Union, concat, *, reversal, substitution, homomorphism and inverse homomorphism, INIT, LAST, MID, EXTERIOR, quotient with regular.
- Decision algorithms: Empty, finite, infinite; CKY for membership.

© UCF (Charles E. Hughes)

Assignment # 1

Assignment #1

Take Home Review
This is a review of COT 4210 material. It serves as a wake up call if you are not familiar with the material and as a gauge for me.

Due: October 15

Computability

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

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 of 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, Matiyacevic showed undecidable; Martin Davis et al. contributed key ideas to showing this).

Hilbert's Belief

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

 © UCF (Charles E. Hughes)

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

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) # 1

- **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".

Categorizing Problems (Sets) # 2

- **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), Non-re, Not Semi-Decidable** -- Complements of …

Immediate Implications

- **P enumerable iff P semi-decidable.**
- **P solvable iff both S_P and $(U — S_P)$ are re (semi-decidable).**
- **P solved implies P solvable implies P semi-decidable (re).**
- **P non-re implies P unsolvable implies P unsolved.**
- **P finite implies P solvable.**

© UCF (Charles E. Hughes)

Goals of Computability (Again)

- **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.**

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 in class

The Need for Divergence

For vs While Loops

Bounded Iteration

- **Any programming language that limits iteration to control structures in which we can pre-compute a bound on the number of repetitions is an incomplete language.**
- **In other words, the possibility of divergence is essential to a complete model of effective computation.**
- **I will prove this in class, along with showing you Cantor's proof that there are more reals in [0,1) than there are natural numbers.**

Hilbert's Tenth

Diophantine Equations are Semi-decidable

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

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

P(x) = 0 is Decidable

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

© UCF (Charles E. Hughes)

Models of Computation

***S*-Programs**
Register Machines
Factor Replacement Systems
Recursive Functions
Turing Machines

S-Programs

1st Model

A Familiar Feeling Number Manipulation Language

S-Program Concept

- **An *S*-program consists of a finite length program, each of whose instructions is chosen from a small repertoire of simple commands.**
- **The instructions are optionally labeled with symbolic tags used in branching commands. Termination occurs as a result of an attempt to branch to a non-existent label.**

S Program Basic parts

- **Computation is limited to non-negative values. These are stored in a set of variables.**
- **The inputs for an n-ary function are in variables X1, X2, … , Xn; output is in Y; and Z1, Z2, … are available for storing intermediate results.**
- **Y and all Z-variables start with the value zero.**
- **Labels are chosen from A1, B1, C1, D1, E1, A2, B2, C2, D2, E2, …**
- **Simple names X, Z, A, B, C, D and E are often used in place of X1, Z1, A1, B1, C1, D1 and E1, respectively.**

Primitive *S* Commands

- **The primitive commands are (the labels are optional):**

  [A] $V \leftarrow V + 1$

  [B] $V \leftarrow V - 1$

  [C] $V \leftarrow V$

  [D] IF $V \neq 0$ GOTO L

© UCF (Charles E. Hughes)

Useful Macros # 1

GOTO A

$Z \leftarrow Z+1$

IF $Z \neq 0$ GOTO A

IF V = 0 GOTO A

IF $V \neq 0$ GOTO E

GOTO A

[E] …

© UCF (Charles E. Hughes)

Useful Macros # 2

$V \leftarrow 0$

[A] IF V = 0 GOTO E

V ← V-1

GOTO A

[E] …

$V \leftarrow k+1$ // assume we have macro for $V \leftarrow k$

V ← k

V ← V+1

Useful Macros # 3

$V \leftarrow U$

```
[A]  IF U = 0 GOTO B
     U ← U-1
     Z ← Z+1
     GOTO A
[B]  V ← 0
[C]  IF Z = 0 GOTO E
     Z ← Z-1
     U ← U+1
     V ← V+1
     GOTO C
[E]  …
```


© UCF (Charles E. Hughes)

Addition by *S* Program

Compute V + U (args are X1, X2)

```
      Z1 ← X1
      Z2 ← X2
  [A] IF Z2 = 0 GOTO B
      Z1 ← Z1+1
      Z2 ← Z2-1
      GOTO A
  [B] Y ← Z1
      Z1 ← 0
```


© UCF (Charles E. Hughes)

Subtraction

Compute V – U, if V≥U; ↑, otherwise (args are X1, X2)

```
      Z1 ← X1
      Z2 ← X2
[A]   IF Z2 = 0 GOTO B
      IF Z1 = 0 GOTO A
      Z1 ← Z1-1
      Z2 ← Z2-1
      GOTO A
[B]   Y ← Z1
      Z1 ← 0
```


© UCF (Charles E. Hughes)

Limited Subtraction

Compute V – U, if V≥U; 0, otherwise (args are X1, X2)

```
     Z1 ← X1
     Z2 ← X2
[A]  IF Z1 = 0 GOTO B
     IF Z2 = 0 GOTO C
     Z1 ← Z1-1
     Z2 ← Z2-1
     GOTO A
[B]  Z2 ← 0
[C]  Y ← Z1
     Z1 ← 0
```

Alternative Version

Compute V – U, if V≥U; 0, otherwise (args are X1, X2)

```
      Z1 ← X1
      Z2 ← X2
[A]   IF Z2 = 0 GOTO C
      Z1 ← Z1-1
      Z2 ← Z2-1
      GOTO A
[C]   Y ← Z1
      Z1 ← 0
```

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, I will not use subscripts if obvious.**

© UCF (Charles E. Hughes)

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 $\leftarrow$ 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 (Charles E. Hughes)

Addition by FRS

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

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

2 / 3

2 / 5

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

$bx \rightarrow ax$

meaning that a number that has a factored as bx can have the factor b replaced by an a.
The previous rules would then be written

$3x \rightarrow 2x$

$5x \rightarrow 2x$

Limited Subtraction by FRS

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

$3 \cdot 5x \to x$

$3x \to 2x$

$5x \to x$

Ordering of Rules

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

Why Deterministic?

To see why determinism makes a difference, consider

$3 \cdot 5x \rightarrow x$

$3x \rightarrow 2x$

$5x \rightarrow x$

Starting with $135 = 3^3 5^1$, 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

$$2x \rightarrow x$$

$$2x \rightarrow 4x$$

starting with the number 2.

Systems Related to FRS

- **Petri Nets:**
  - Unordered
  - Ordered
  - Negated Arcs
- **Vector Addition Systems:**
  - Unordered
  - Ordered
- **Factors with Residues:**
  - $a\ x + c \quad \rightarrow \quad b\ x + d$
- **Finitely Presented Abelian Semi-Groups**

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.**

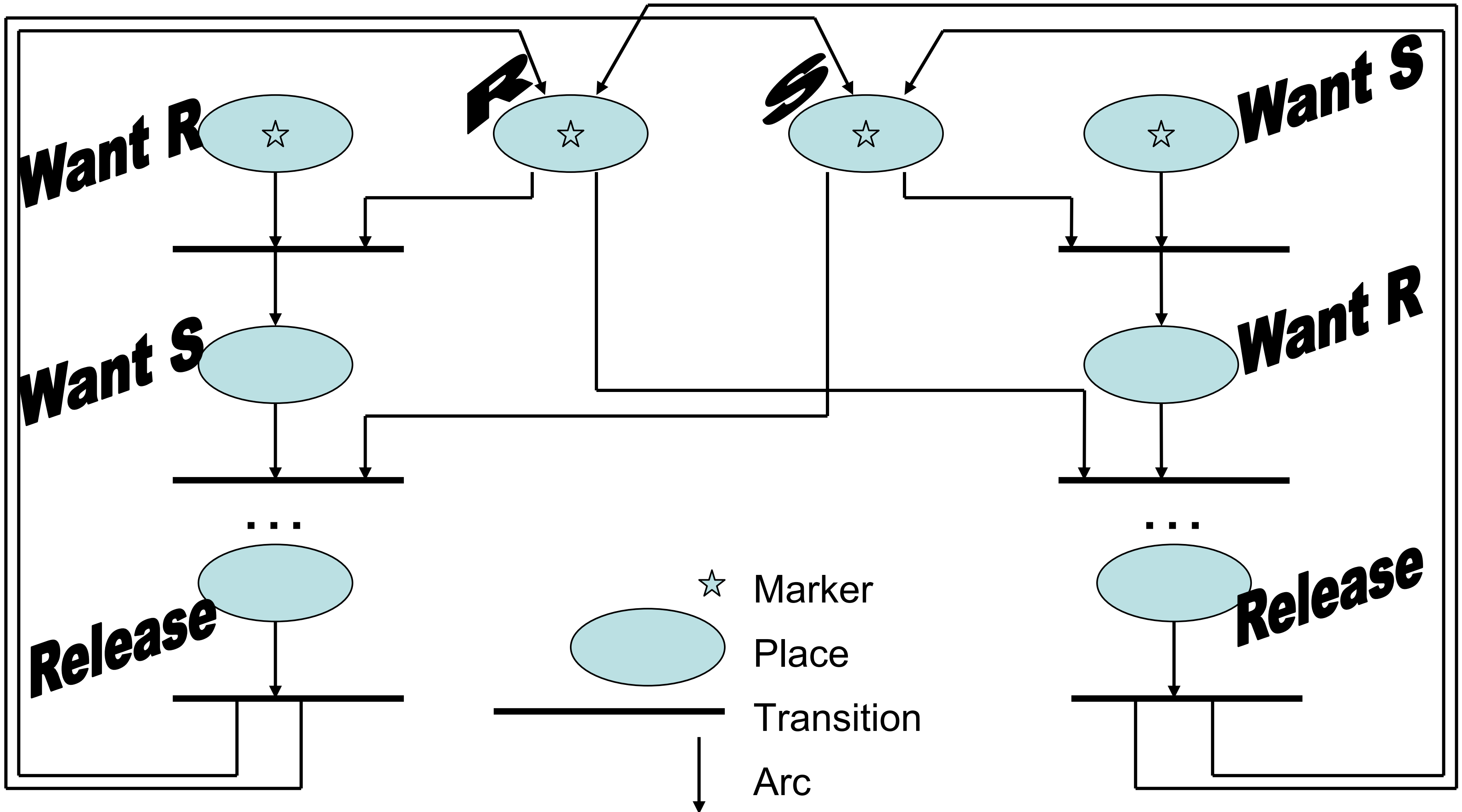
Petri Net Computation

- **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.**

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.

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.**

 © UCF (Charles E. Hughes)

Abelian Semi-Group

S = (G, •) is a semi-group if

G is a set, • is a binary operator, and

1. Closure: If $x,y \in G$ then $x \bullet y \in G$
2. Associativity: $x \bullet (y \bullet z) = (x \bullet y) \bullet z$

S is a monoid if

3. Identity: $\exists e \in G\ \forall x \in G\ [e \bullet x = x \bullet e = x]$

S is a group if

4. Inverse: $\forall x \in G\ \exists x^{-1} \in G\ [x^{-1} \bullet x = x \bullet x^{-1} = e]$

S is Abelian if • is commutative

Finitely Presented

- **S = (G, •), a semi-group (monoid, group), is finitely presented if there is a finite set of symbols, Σ, called the alphabet or generators, and a finite set of equalities ($\alpha_i = \beta_i$), the reflexive transitive closure of which determines equivalence classes over G.**
- **Note, the set G is the closure of the generators under the semi-group's operator •.**
- **The problem of determining membership in equivalence classes for finitely presented Abelian semi-groups is equivalent to that of determining mutual derivability in an unordered FRS or Vector Addition System with inverses for each rule.**

Assignment # 2

a. **Present a Register Machine that computes FIB. Assume R1=x; at termination, set R2=1 if x is a member of the Fibonacci sequence and 0 if not.**

b. **Present a Factor Replacement System that computes FIB. Assume starting number is 3^x 5; at termination, result is 2=2^1 if x is a member of the Fibonacci sequence; 1= 2^0 otherwise. Actually, it can be done without the 5, but that may make it easier.**

c. **Prove that non-deterministic FRS's are no more powerful than non-deterministic VAS. This means you need only show that any non-deterministic FRS can be simulated by a non-deterministic VAS.**

   Note: To do this most effectively, you need to first develop the notion of an instantaneous description (ID) of a FRS (that's a point in 1-space) and of a VAS (that's a point in n-space). You then need a mapping from an FRS ID to a corresponding VAS ID, and this mapping needs to be some function (many-one into), f. Next, there must be a mapping from the rules of the FRS to create those of the VAS, such that a single step of the FRS from x to y is mimicked by some finite number of steps of the VAS from f(x) to f(y), where f(y) is the first ID derived from f(x) that is a mapping from some ID of the VAS

Due: September 10

Recursive Functions

Primitive and μ-Recursive

Primitive Recursive

An Incomplete Model

Basis of PRFs

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

$C_a(x_1,\ldots,x_n) = a$	: constant functions
$I^n_i(x_1,\ldots,x_n) = x_i$	: identity functions
	: aka projection
$S(x) = x+1$	: an increment function

Building New Functions

- **Composition:**

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

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

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

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

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

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

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

Addition & Multiplication

Example: Addition

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

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

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

Example: Multiplication

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

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

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

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

Basic Arithmetic

x + 1:

$x + 1 = S(x)$

x – 1:

$0 - 1 = 0$

$(x+1) - 1 = x$

x + y:

$x + 0 = x$

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

x – y: // limited subtraction

$x - 0 = x$

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

2nd Grade Arithmetic

x * y:

x * 0 = 0

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

x!:

0! = 1

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

Basic Relations

x == 0:

0 == 0 = 1

(y+1) == 0 = 0

x == y:

x==y = ((x – y) + (y – x)) == 0

x ≤y :

x≤y = (x – y) == 0

x ≥ y:

x≥y = y≤x

x > y :

x>y = ~(x≤y) /* See ~ on next page */

x < y :

x<y = ~(x≥y)

Basic Boolean Operations

~x:

    ~x = 1 – x or (x==0)

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

    ~(x==0)

x && y:

    x&&y = signum(x*y)

x || y:

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

© UCF (Charles E. Hughes)

Definition by Cases

One case

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

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

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

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

Bounded Minimization 1

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

$= x+1$, otherwise

where P(z) is primitive recursive.

Can show f is primitive recursive by

$f(0)$	=	$1-P(0)$	
$f(x+1)$	=	$f(x)$	if $f(x) \leq x$
	=	$x+2-P(x+1)$	otherwise

© UCF (Charles E. Hughes)

Bounded Minimization 2

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

$\quad = x$, otherwise

where P(z) is primitive recursive.

Can show f is primitive recursive by

$f(0) = 0$

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

Intermediate Arithmetic

x // y:

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

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

x | y: x is a divisor of y

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

© UCF (Charles E. Hughes)

Primality

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

firstfactor(x) = μ z (2 ≤ z ≤ x) [z|x] ,
0 if none

isPrime(x):

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

prime(i) = i-th prime:

prime(0) = 2

prime(x+1) = μ z(prime(x)< z ≤prime(x)!+1)[isPrime(z)]

We will abbreviate this as p_i for prime(i)

Exponents

x^y:

$x^0 = 1$

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

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

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

© UCF (Charles E. Hughes)

Pairing Functions

- pair(x,y) = $\langle x,y \rangle = 2^x (2y + 1) - 1$
- with inverses

  $\langle z \rangle_1 = \exp(z+1,0)$

  $\langle z \rangle_2 = (((z + 1) // 2^{\langle z \rangle_1}) - 1) // 2$
- These are very useful and can be extended to encode n-tuples

  $\langle x,y,z \rangle = \langle x, \langle y,z \rangle \rangle$ (note: stack analogy)

Assignment # 3

Show that prfs are closed under mutual recursion. That is, assuming F1, F2 and G1 and G2 are pr, show that H1 and H2 are, where

H1(0, x) = F1(x); H2(0, x) = F2(x)

H1(y+1, x) = G1(y,x,H2(y,x)); H2(y+1, x) = G2(y,x,H1(y,x))

Hint: The pairing function is useful here.

Due: September 17

© UCF (Charles E. Hughes)

μ Recursive

4th Model

A Simple Extension to Primitive Recursive

μ Recursive Concepts

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

Ackermann's Function

- $A(1, j)=2j$ for $j \geq 1$
- $A(i, 1)=A(i-1, 2)$ for $i \geq 2$
- $A(i, j)=A(i-1, A(i, j-1))$ for $i, j \geq 2$
- Wilhelm Ackermann observed in 1928 that this is not a primitive recursive function.
- Ackermann's function grows too fast to have a for-loop implementation.
- The inverse of Ackermann's function is important to analyze Union/Find algorithm.

 © UCF (Charles E. Hughes)

Union/Find

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

The μ Operator

- **Minimization:**

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

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

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

Turing Machines

5th 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.**

 © UCF (Charles E. Hughes)

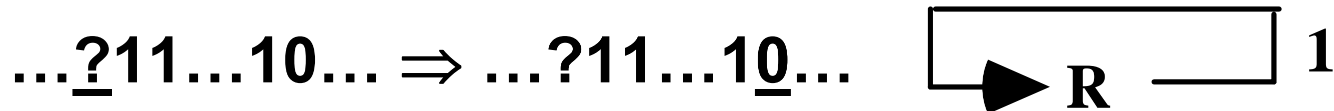
Base Machines

- **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

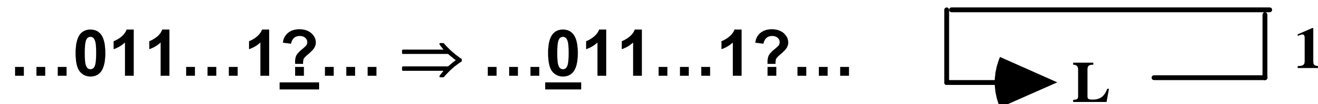
$\mathcal{R}$ -- move right to next 0 (not including current square)

…?11…10… ⇒ …?11…10…



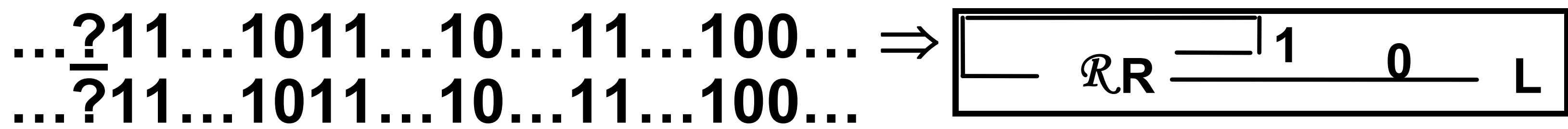
$\mathcal{L}$ -- move left to next 0 (not including current square)

…011…1?… ⇒ …011…1?…



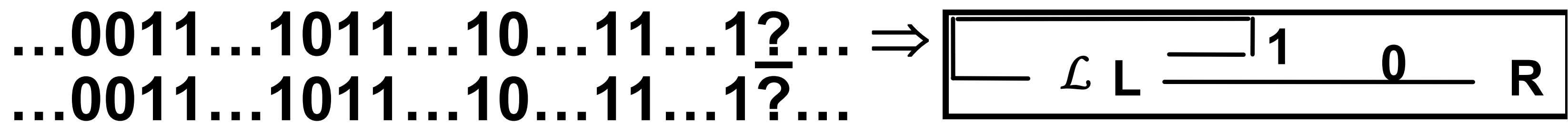
$\mathfrak{R}$ -- move right to next 00 (not including current square)

…?11…1011…10…11…100… ⇒
…?11…1011…10…11…100…



$\mathfrak{L}$ -- move left to next 00 (not including current square)

…0011…1011…10…11…1?… ⇒
…0011…1011…10…11…1?…



© UCF (Charles E. Hughes)

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.

© UCF (Charles E. Hughes)

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

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

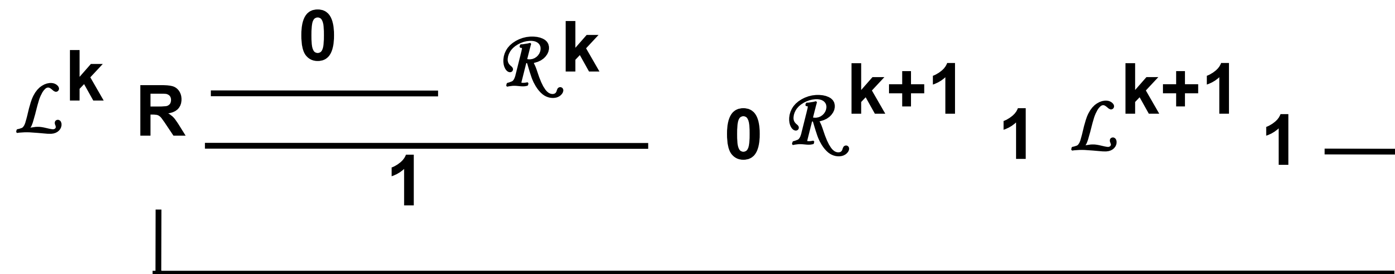
and ends with

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

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

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

© UCF (Charles E. Hughes)

Turing Machine Variations

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

© UCF (Charles E. Hughes)

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.**
- **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.**

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.**

© UCF (Charles E. Hughes)

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

- **$\varphi_{M,w} = \phi cell \wedge \phi start \wedge \phi move \wedge \phi accept$**
- **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**

Equivalence of Models

Equivalency of computation by *S*-programs, register machines, factor replacement systems, recursive functions and Turing machines

S-Machine $\equiv$ REGISTER

S Program ≤ Reg. Machine

- Let P be an *S* Program consisting of m instructions computing f(x1,…, xn). Assume the highest indexed temporary variable is Z_t
- Define the mapping g, $g(X_i) = i$, $1 \le i \le n$, $g(Y) = n+1$, and $g(Z_j) = n+j+1$, $1 \le j \le t$.
- Change each IF V≠0 GOTO L to IF V≠0 GOTO A_k, where L is the k-th instruction, or if L is E, k=m+1
- Map the j-th *S* instruction by
  - $[A_j]$ V ← V maps to
    - 2j-1. $DEC_{n+t+2}(2j,2j)$
    - 2j. $DEC_{n+t+2}(2j+1,2j+1)$
  - $[A_j]$ V ← V+1 maps to
    - 2j-1. $INC_{g(v)}(2j)$
    - 2j. $DEC_{n+t+2}(2j+1,2j+1)$
  - $[A_j]$ V ← V-1 maps to
    - 2j-1. $DEC_{g(v)}(2j,2j)$
    - 2j. $DEC_{n+t+2}(2j+1,2j+1)$
  - $[A_j]$ IF V≠0 GOTO A_k maps to
    - 2j-1. $DEC_{g(v)}(2j,2j+1)$
    - 2j. $INC_{g(v)}(2k-1)$

Reg. Machine ≤ *S* Program

- **Let M be a Register Machine consisting of m instructions computing $f(x1,\ldots, xn)$. Assume highest indexed register is R_s**
- **Define the mapping g, $g(i)=X_i$, $1\le i\le n$, $g(n+1)=Y$, and $g(i)=Z_{i-n-1}$, $n+2\le i\le s$.**
- **Start the *S* Program with the command**
  - **$[A_1]$ $Z_{s-n} \leftarrow Z_{s-n} + 1$**
- **Map the j-th Register Machine instruction by**
  - **j. $INC_r(k)$ to**
    $[A_{4j-2}]$ $g(r) \leftarrow g(r) + 1$
    $[A_{4j-1}]$ IF $g(r)\neq 0$ GOTO A_{4k-2}
    $[A_{4j}]$ $g(r) \leftarrow g(r)$
    $[A_{4j+1}]$ $g(r) \leftarrow g(r)$
  - **j. $DEC_r(p,z)$ to**
    $[A_{4j-2}]$ IF $g(r)\neq 0$ GOTO A_{4j}
    $[A_{4j-1}]$ IF $Z_{s-n}\neq 0$ GOTO A_{4z-2}
    $[A_{4j}]$ $g(r) \leftarrow g(r) - 1$
    $[A_{4j+1}]$ IF $Z_{s-n}\neq 0$ GOTO A_{4p-2}
- **The 4m+1 instructions above are ordered by their labels. Note that label A_{4m+2} may be recast as the special label E.**

Proving Equivalence

- **The previous constructions do not, by themselves, prove equivalence.**
- **To do so, we need to develop a notion of an "instantaneous description" (id) of an S-program and of a register machine.**
- **We will then show a mapping of id's between the models.**

© UCF (Charles E. Hughes)

Instantaneous Descriptions

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

id Definitions

- **For an *S*-program, P, an id is an n+t+2 tuple of the form $(i, x_1,\ldots,x_n, y, z_1,\ldots,z_t)_P$ specifying the number of the next instruction to be executed and the values of all variables prior to its execution.**
- **For a register machine, M, an id is an s+1 tuple of the form $(i, r_1,\ldots,r_s)_M$ specifying the number of the next instruction to be executed and the values of all registers prior to its execution.**

Equivalence Steps

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

Completion of S-P ≤ RM

- **To go from S-program, P, to reg. machine, M, define $h(i,x_1,\ldots,x_n,y,z_1,\ldots,z_t)_P=(2i-1,x_1,\ldots,x_n,y,z_1,\ldots,z_t,0)_M$ under our previous association of $x_1,\ldots,x_n$ with $r_1,\ldots,r_n$, y with r_{n+1}, and $z_1,\ldots,z_t$ with $r_{n+2},\ldots,r_{n+1+t}$.**
- **The proof can now be completed as follows.**
  - Note that, when computing $f(a_1,\ldots,a_n)$, P starts on $id_0 = (1,a_1,\ldots,a_n,0,0\ldots,0)_P$ and M starts at $h(id_0) = (1, a_1,\ldots,a_n,0,0\ldots,0,0)_M$.
  - Show that that our instruction mappings preserve the h-mapping, above. This requires a simple case analysis for the four S-program instruction types. Since this takes two steps per instruction, you must note that no intermediary id is in the range of h, but that's easy as they have even instruction counters.

Completion of RM ≤ S-P

- **To go from reg. machine, M, to S-program, P, define $h(i,r_1,\ldots,r_n,r_{n+1},r_{n+2},\ldots,r_s)_M=(4i\text{-}2,r_1,\ldots,r_n,r_{n+1},r_{n+2},\ldots,r_s,1)_P$ under our previous association of $x_1,\ldots,x_n$ with $r_1,\ldots,r_n$, y with r_{n+1}, and $z_1,\ldots,z_{s-n-1}$ with $r_{n+2},\ldots,r_s$.**
- **The proof can now be completed as follows.**
  - Note that, when computing $f(a_1,\ldots,a_n)$,
    M starts on $id_0 = (1,a_1,\ldots,a_n,0,0\ldots,0)_M$ and
    P starts at $(1, a_1,\ldots,a_n,0,0\ldots,0,0)_P$ and in one step transitions to $h(id_0) = (2, a_1,\ldots,a_n,0,0\ldots,0,1)_P$.
  - Show that that our instruction mappings preserve the h-mapping, above. This requires a simple case analysis for the two register machine instruction types. Since this takes more than one step per instruction, you must note that no intermediary id is in the range of h.

© UCF (Charles E. Hughes)

All Models are Equivalent

Our Plan of Attack

- **We will now show**
  TURING ≤ REGISTER ≤ FACTOR ≤
  RECURSIVE ≤ TURING
  where by A ≤ B, we mean that every instance of A can be replaced by an equivalent instance of B.
- **The transitive closure will then get us the desired result.**

TURING $\leq$ REGISTER

Encoding a TM's State

- **Assume that we have an n state Turing machine. Let the states be numbered 0,…, n-1.**
- **Assume our machine is in state 7, with its tape containing**
  … 0 0 1 0 1 0 0 1 1 q7 <u>0</u> 0 0 …
- **The underscore indicates the square being read. We denote this by the finite id**
  1 0 1 0 0 1 1 q7 <u>0</u>
- **In this notation, we always write down the scanned square, even if it and all symbols to its right are blank.**

More on Encoding of TM

- An id can be represented by a triple of natural numbers, (R,L,i), where R is the number denoted by the reversal of the binary sequence to the right of the qi, L is the number denoted by the binary sequence to the left, and i is the state index.
- So,
  … 0 0 1 0 1 0 0 1 1 q7 0 0 0 …
  is just (0, 83, 7).
  … 0 0 1 0 q5 1 0 1 1 0 0 …
  is represented as (13, 2, 5).
- We can store the R part in register 1, the L part in register 2, and the state index in register 3.

Simulation by RM

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

Fixups

- **Need epilog so action for missing quad (halting) jumps beyond end of simulation to clean things up, placing result in r1.**
- **Can also have a prolog that starts with arguments in first n registers and stores values in r1, r2 and r3 to represent Turing machines starting configuration.**

© UCF (Charles E. Hughes)

Prolog

Example assuming n arguments (fix as needed)

1.	**MUL_rn+1_BY_2[2]**	**: Set rn+1 = $11\ldots10_2$, where, #1's = r1**
2.	**DEC1[3,4]**	**: r1 will be set to 0**
3.	**INCn+1[1]**	**:**
4.	**MUL_rn+1_BY_2[5]**	**: Set rn+1 = $11\ldots1011\ldots10_2$, where, #1's = r1, then r2**
5.	**DEC2[6,7]**	**: r2 will be set to 0**
6.	**INCn+1[4]**	**:**
…		
3n-2.	**DECn[3n-1,3n+1]**	**: Set rn+1 = $11\ldots1011\ldots1011\ldots1_2$, where, #1's = r1, r2,…**
3n-1.	**MUL_rn+1_BY_2[3n]**	**: rn will be set to 0**
3n.	**INCn+1[3n-2]**	**:**
3n+1	**DECn+1[3n+2,3n+3]**	**: Copy rn+1 to r1, rn+1 is set to 0**
3n+2.	**INC2[3n+1]**	**:**
3n+3.		**: r2 = left tape, r1 = 0 (right), r3 = 0 (initial state)**

Epilog

1. DEC3[1,2] : Set r3 to 0 (just cleaning up)
2. IF_r1_ODD[3,5] : Are we done with answer?
3. INC2[4] : putting answer in r2
4. DIV_r1_BY_2[2] : strip a 1 from r1
5. DEC1[5,6] : Set r1 to 0 (prepare for answer)
6. DEC2[6,7] : Copy r2 to r1
7. INC1[6] :
8. : Answer is now in r1

© UCF (Charles E. Hughes)

REGISTER $\leq$ FACTOR

Encoding a RM's State

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

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

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

- **Let R be an arbitrary n-register machine, having m instructions.**

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

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

  Use pn+m+1 as prime factor that indicates simulation is done.

- **This is in essence the Gödel number of the RM's state.**

Simulation by FRS

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

j. INCr[i]

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

j. DECr[s, f]

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

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

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

$$p_{n+m+1}x \rightarrow x$$

Importance of Order

- The relative order of the two rules to simulate a DEC are critical.
- To test if register r has a zero in it, we, in effect, make sure that we cannot execute the rule that is enabled when the r-th prime is a factor.
- If the rules were placed in the wrong order, or if they weren't prioritized, we would be non-deterministic.

Example of Order

Consider the simple machine to compute r1:=r2 – r3 (limited)

1. **DEC3[2,3]**
2. **DEC2[1,1]**
3. **DEC2[4,5]**
4. **INC1[3]**
5.

© UCF (Charles E. Hughes)

Subtraction Encoding

Start with 3^x5^y7

$7 \bullet 5\ x$	$\rightarrow$	$11\ x$
$7\ x$	$\rightarrow$	$13\ x$
$11 \bullet 3\ x$	$\rightarrow$	$7\ x$
$11\ x$	$\rightarrow$	$7\ x$
$13 \bullet 3\ x$	$\rightarrow$	$17\ x$
$13\ x$	$\rightarrow$	$19\ x$
$17\ x$	$\rightarrow$	$13 \bullet 2\ x$
$19\ x$	$\rightarrow$	x

© UCF (Charles E. Hughes)

Analysis of Problem

- **If we don't obey the ordering here, we could take an input like $3^5 5^2 7$ and immediately apply the second rule (the one that mimics a failed decrement).**
- **We then have $3^5 5^2 13$, signifying that we will mimic instruction number 3, never having subtracted the 2 from 5.**
- **Now, we mimic copying r2 to r1 and get $2^5 5^2 19$.**
- **We then remove the 19 and have the wrong answer.**

© UCF (Charles E. Hughes)

FACTOR $\leq$ RECURSIVE

Universal Machine

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

© UCF (Charles E. Hughes)

Encoding FRS

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

  $a_i x \rightarrow b_i x$

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

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

Simulation by Recursive # 1

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

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

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

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

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

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

Simulation by Recursive # 2

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

CONFIG(F, x, 0) = x

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

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

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

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

© UCF (Charles E. Hughes)

Simulation by Recursive # 3

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

  Univ(F, x) = exp (CONFIG (F, x, HALT (F, x)), 0)

- **This assumes that the answer will be returned as the exponent of the only even prime, 2. We can fix F for any given Factor System that we wish to simulate.**

© UCF (Charles E. Hughes)

Simplicity of Universal

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

  $$F(x) = G(\mu y\ H(x, y))$$

  where G and H are primitive recursive.

RECURSIVE $\leq$ TURING

Standard Turing Computation

- **Our notion of standard Turing computability of some n-ary function F assumes that the machine starts with a tape containing the n inputs, x1, … , xn in the form**

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

  and ends with

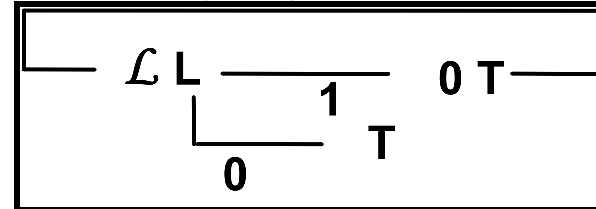
  $\ldots 0 1^{x1} 0 1^{x2} 0 \ldots 0 1^{xn} 0 1^{y} \underline{0} \ldots$

  where y = F(x1, … , xn).

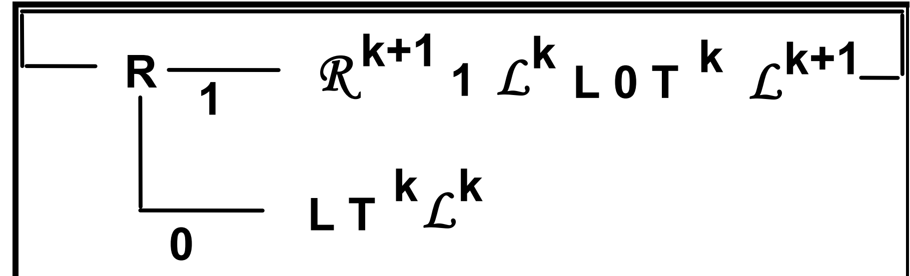
More Helpers

- To build our simulation we need to construct some useful submachines, in addition to the $\mathcal{R}$, $\mathcal{L}$, R, L, and C_k machines already defined.
- T -- translate moves a value left one tape square
  …$\underline{?}01^x0$… ⇒ …$?1^x\underline{0}0$…

- Shift -- shift a rightmost value left, destroying value to its left
  …$01^{x1}01^{x2}\underline{0}$… ⇒ …$01^{x2}\underline{0}$…



- Rot_k -- Rotate a k value sequence one slot to the left
  …$\underline{0}1^{x1}01^{x2}0$…$01^{xk}0$…
  ⇒ …$\underline{0}1^{x2}0$…$01^{xk}01^{x1}0$…



Basic Functions

All Basis Recursive Functions are Turing computable:

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

  $(R1)^a R$

- $I_i^n(x_1,\ldots,x_n) = x_i$

  C_{n-i+1}

- $S(x) = x+1$

  $C_1 1R$

Closure Under Composition

If G, H_1, … , H_k are already known to be Turing computable, then so is F, where

$F(x_1,\ldots,x_n) = G(H1(x_1,\ldots,x_n), \ldots , Hk(x_1,\ldots,x_n))$

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

$E\langle m\rangle(x_1,\ldots,x_n, y_1,\ldots,y_m) = E(x_1,\ldots,x_n)$

This can be computed by the machine

$\mathcal{L}^{n+m} (Rot_{n+m})^n \mathcal{R}^{n+m} E \; \mathcal{L}^{n+m+1} (Rot_{n+m})^m \mathcal{R}^{n+m+1}$

Can now define F by

$H_1 \; H_2\langle 1\rangle \; H_3\langle 2\rangle \ldots H_k\langle k-1\rangle \; G \; Shift^k$

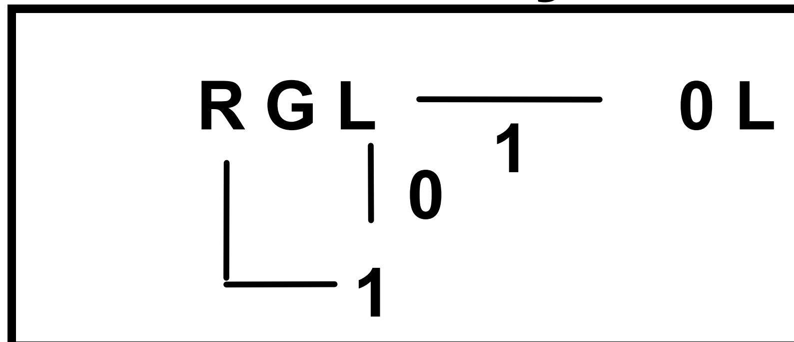
 © UCF (Charles E. Hughes)

Closure Under Minimization

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

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

This can be done by



Assignment # 4

a. **Present a Turing Machine to do MAX of n non-zero arguments, n>=0. You know you've run out of arguments when you encounter the value 0, represented by two successive 0's (blanks). Use the machines we have already built up and others you build. Do NOT turn in Turing Tables. We won't pay any attention to them if you do.**

b. **Show that Turing Machine are closed under iteration (primitive recursion). This completes the equivalence proofs for our five models of computation.**

c. **Constructively (no proof required), show how a standard register machine can simulate a different register machine model with instructions of form:**

   i. if even(r) goto j // goto j if value in register r is even

   i. r = r+1 // increment contents of r

   i. r = r-1 // decrement contents of r

   Note: all registers except input ones start with 0; inputs are in registers r_1, r_2,…, r_n; output in r_{n+1}

Due: September 24

Consequences of Equivalence

- **Theorem: The computational power of S-Programs, Recursive Functions, Turing Machines, Register Machine, and Factor Replacement Systems are all equivalent.**
- **Theorem: Every Recursive Function (Turing Computable Function, etc.) can be performed with just one unbounded type of iteration.**
- **Theorem: Universal machines can be constructed for each of our formal models of computation.**

Undecidability

We Can't Do It All

Undecidability Precursor

- **We can see that there are undecidable functions merely by noting that there are an uncountable number of mappings from the natural numbers into the natural numbers. Since effective procedures are always over a language with a finite number of primitives, and since we restrict programs to finite length, there can be only a countable number of effective procedures. Thus no formalism can get us all mappings -- some must be non-computable.**
- **The above is a great existence proof, but is unappealing since it doesn't help us to understand what kinds of problems are uncomputable. The classic unsolvable problem is called the Halting Problem. It is the problem to decide of an arbitrary effective procedure f: $\aleph \rightarrow \aleph$, and an arbitrary n $\in \aleph$, whether or not f(n) is defined.**

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 ℵ into ℵ 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

Disagree(x) =
- **0** if Halt(x,x) = 0, i.e, if [x] (x) is not defined
- **μy (y == y+1)** if Halt(x,x) = 1, i.e, if [x] (x) is defined

Since Disagree is a program from ℵ into ℵ , 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.

Additional Notations

Includes comment on our notation versus that of others

Universal Machine

- **Others consider functions of *n* arguments, whereas we had just one. However, our input to the FRS was actually an encoding of n arguments.**
- **The fact that we can focus on just a single number that is the encoding of n arguments is easy to justify based on the pairing function.**
- **Some presentations order arguments differently, starting with the *n* arguments and then the Gödel number of the function, but closure under argument permutation follows from closure under substitution.**

Universal Machine Mapping

- $\Phi^{(n)}(x_1,\ldots,x_n, f) = \text{Univ}\ (f, \prod_{i=1}^{n} p_i^{x_i})$
- We will sometimes adopt the above and also its common shorthand

  $\Phi_f^{(n)}(x_1,\ldots,x_n) = \Phi^{(n)}(x_1,\ldots,x_n, f)$

  and the even shorter version

  $\Phi_f(x_1,\ldots,x_n) = \Phi^{(n)}(x_1,\ldots,x_n, f)$

SNAP and TERM

- **Our CONFIG is essentially the common SNAP (snapshot) with arguments permuted**

  SNAP(x, f, t) = CONFIG(f, x, t)

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

  TERM(x, f) = (NEXT(f, x) == x)

- **Again, we used a single argument but that can be extended as we have already shown.**

STP Predicate

- **STP(x1,…,xn, f, t) is a predicate defined to be true iff [f](x1,…,xn) converges in at most t steps.**
- **STP is primitive recursive since it can be defined by**

  STP(x, f, *s*) = TERM(CONFIG(f, x, *s*), f)

  Extending to many arguments is easily done as before.

Recursively Enumerable

Properties of re Sets

Definition of re

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

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

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

- **I prefer the definition of re that says**
  $S \subseteq \aleph$ is re iff $S = \emptyset$ or there exists a totally computable function f where

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

- **We will prove these equivalent. Actually, f can be a primitive recursive function.**

© UCF (Charles E. Hughes)

Semi-Decidable Implies re

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

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

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

re Implies Semi-Decidable

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

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

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

© UCF (Charles E. Hughes)

Domain of a Procedure

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

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

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

Recursive Implies re

Theorem: Recursive implies re.

Proof: S is recursive implies there is a total recursive function f_S such that

$$S = \{ x \in \aleph \mid f_s(x) == 1 \}$$

Define $g_s(x) = \mu y\ (f_s(x) == 1)$

Clearly

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

Related Results

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

Proof: That's what we proved.

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

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

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

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

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

© UCF (Charles E. Hughes)

Enumeration Theorem

- Define
  $W_n = \{ x \in \aleph \mid \Phi(x,n)\downarrow \}$
- Theorem: A set B is re iff there exists an n such that $B = W_n$.
  Proof: Follows from definition of $\Phi(x,n)$.
- This gives us a way to enumerate the recursively enumerable sets.
- Note: We will later show (again) that we cannot enumerate the recursive sets.

 © UCF (Charles E. Hughes)

The Set K

- **$K = \{ n \in \aleph \mid n \in W_n \}$**
- **Note that**
  $n \in W_n \Leftrightarrow \Phi(n,n)\downarrow \Leftrightarrow HALT(n,n)$
- **Thus, K is the set consisting of the indices of each program that halts when given its own index**
- **K can be semi-decided by the HALT predicate above, so it is re.**

K is not Recursive

- **Theorem: We can prove this by showing ~K is not re.**
- **If ~K is re then ~K = W_i, for some i.**
- **However, this is a contradiction since $i \in K \Leftrightarrow i \in W_i \Leftrightarrow i \in \sim K \Leftrightarrow i \notin K$**

© UCF (Charles E. Hughes)

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 (Charles E. Hughes)

S-m-n Theorem

Parameter (S-m-n) Theorem

- **Theorem: For each n,m>0, there is a prf $S_m^n(u_1,\ldots,u_n,y)$ such that**

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

- **The proof of this is highly dependent on the system in which you proved universality and the encoding you chose.**

S-m-n for FRS

- **We would need to create a new FRS, from an existing one F, that fixes the value of u_i as the exponent of the prime p_{m+i}.**
- **Sketch of proof:**
  Assume we normally start with $p_1^{x1} \dots p_m^{xm}\ p_1^{u1} \dots p_{m+n}^{un}\ \sigma$
  Here the first m are variable; the next n are fixed; σ denotes prime factors used to trigger first phase of computation.
  Assume that we use fixed point as convergence.
  We start with just $p_1^{x1} \dots p_m^{xm}$, with q the first unused prime.

  $q\ \alpha\ x \rightarrow q\ \beta\ x$ replaces $\alpha\ x \rightarrow \beta\ x$ in F
  $q\ x \rightarrow q\ x$ ensures we loop at end
  $x \rightarrow q\ p_{m+1}^{u1} \dots p_{m+n}^{un}\ \sigma\ x$
  adds fixed input, start state and q
  this is selected once and never again

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

Details of S-m-n for FRS

- The number of F (called F, also) is $2^r 3^{a_1} 5^{b_1} \dots p_{2r-1}^{a_r} p_{2r}^{b_r}$
- $S_{m,n}(u_1,\dots u_n, F) = 2^{r+2} 3^{q\times a_1} 5^{q\times b_1} \dots p_{2r-1}^{q\times a_r} p_{2r}^{q\times b_r}$ $p_{2r+1}^{q} p_{2r+2}^{q} p_{2r+3} p_{2r+4}^{q} p_{m+1}^{u_1} \dots p_{m+n}^{u_n} \sigma$
- This represents the rules we just talked about. The first added rule pair means that if the algorithm does not use fixed point, we force it to do so. The last rule pair is the only one initially enabled and it adds the prime q, the fixed arguments $u_1,\dots u_n$, the enabling prime q, and the σ needed to kick start computation. Note that σ could be a 1, if no kick start is required.
- $S_{m,n}=S_m^n$ is clearly primitive recursive. I'll leave the precise proof of that as a challenge to you.

Quantification#1

- **S is decidable iff there exists an algorithm χ_S (called S's characteristic function) such that $x \in S \Leftrightarrow \chi_S(x)$ This is just the definition of decidable.**
- **S is re iff there exists an algorithm A_S where $x \in S \Leftrightarrow \exists t\ A_S(x,t)$ This is clear since, if g_S is the index of the procedure ψ_S defined earlier that semi-decides S then $x \in S \Leftrightarrow \exists t\ STP(x, g_S, t)$ So, $A_S(x,t) = STP_{g_S}(x, t)$, where STP_{g_S} is the STP function with its second argument fixed.**
- **Creating new functions by setting some one or more arguments to constants is an application of S_m^n.**

Quantification#2

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

Diagonalization and Reducibility

Non-re Problems

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

  $F(x) = F_x$

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

The Contradiction

- **Define** $G(x) = Univ(F(x), x) + 1 = \Phi(x, F(x)) = F_x(x) + 1$
- **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.**

© UCF (Charles E. Hughes)

The Set TOT

- **The listing of all algorithms can be viewed as**

  $TOT = \{ f \in \aleph \mid \forall x\ \Phi(x,f)\downarrow \}$

- **We can also note that**

  $TOT = \{ f \in \aleph \mid W_f = \aleph \}$

- **Theorem: TOT is not re.**

Quantification#3

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

  $f \in TOT \Leftrightarrow \forall x \exists t\ (\ STP(\ x, f, t\ )\ )$

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

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.**

 © UCF (Charles E. Hughes)

Reduction Example

- **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 $y \in \aleph$**
- **Then F_x is an algorithm if and only if F halts on x. This is another application of the S_m^n theorem**
- **Thus a solution to the Uniform Halting Problem would provide a solution to the Halting Problem.**

Classic Undecidable Sets

- **The universal language**

  $K_0 = L_u = \{ <f, x> \mid [f](x) \text{ is defined } \}$

- **Membership problem for L_u is the Halting Problem.**
- **The sets L_{ne} and L_e, where**

  NON-EMPTY = $L_{ne} = \{ f \mid \exists\, x\, [f](x) \text{ is defined } \}$

  EMPTY = $L_e = \{ f \mid \forall\, x\, [f](x) \text{ is undefined } \}$

  are the next ones we will study.

© UCF (Charles E. Hughes)

L_{ne} is re

- L_{ne} is enumerated by

  F(<f, x, t>) = f * STP(x, f, t)

- This assumes that 0 is in L_{ne} since 0 probably encodes some trivial machine. If this isn't so, we'll just slightly vary our enumeration of the recursive functions so it is true.
- Thus, the range of this total function F is exactly the indices of functions that converge for some input, and that's L_{ne}.

L_{ne} is Non-Recursive

- Note in the previous enumeration that F is a function of just one argument, as we are using an extended pairing function $<x,y,z> = <x,<y,z>>$.
- Now L_{ne} cannot be recursive, for if it were then L_u is recursive by the reduction we showed before.
- In particular, from any index x and input y, we created a new function which accepts all input just in case the x-th function accepts y. Hence, this new function's index is in L_{ne} just in case (x, y) is in L_u.
- Thus, a decision procedure for L_{ne} (equivalently for L_e) implies one for L_u.

© UCF (Charles E. Hughes)

L_{ne} is re by Quantification

- **Can do by observing that**

  $f \in L_{ne} \Leftrightarrow \exists <x,t> STP(x, f, t)$

- **By our earlier results, any set whose membership can be described by an existentially quantified recursive predicate is re (semi-decidable).**

L_e is not re

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

$$
\begin{aligned}
f \in L_e \quad & \Leftrightarrow \forall \langle x,t \rangle \sim STP(x, f, t) \\
& \Leftrightarrow \sim\exists \langle x,t \rangle\ STP(x, f, t) \\
& \Leftrightarrow f \notin L_{ne}
\end{aligned}
$$

© UCF (Charles E. Hughes)

Exam#1 Review

You are responsible for the first 196 pages of these notes, except for the P=NP material.

Sample Question#1

1. Present a register machine and a factor replacement system that each produce the value 1 (true), if x>y, and 0 (false), otherwise.
   a) For the register machine, assume it starts with x in R2 and y in R3, and all else 0. The result must end up in R1, with R2 and R3 unchanged.
   b) For the FRS, assume it starts with 3^x5^y and must end up with 2^{result}.

Sample Question#2

2. **Prove that the following are equivalent**
   a) **S is an infinite recursive (decidable) set.**
   b) **S is the range of a monotonically increasing total recursive function. Note: f is monotonically increasing means that $\forall x\ f(x+1) > f(x)$.**

Sample Question#3

3. **Let A and B be re sets. For each of the following, either prove that the set is re, or give a counterexample that results in some known non-re set.**
   a) $A \cup B$
   b) $A \cap B$
   c) $\sim A$

© UCF (Charles E. Hughes)

Sample Question#4

4. **Present a demonstration that the *even* function is primitive recursive.**
   even(x) = 1 if x is even
   even(x) = 0 if x is odd
   You may assume only that the base functions are prf and that prf's are closed under a finite number of applications of composition and primitive recursion.

Sample Question#5

5. Given that the predicate STP is a prf, show that we can semi-decide

   { f | f evaluates to 0 for some input}

   Note: STP(x, f, s) is true iff $\Phi_f(x)$ converges in s or fewer steps

© UCF (Charles E. Hughes)

Sample Question#6

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

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

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

   (a) Can E be non re?

   (b) Can E be re non-recursive?

   (c) Can E be recursive?

Sample Question#7

7. **Assuming that the Uniform Halting Problem is undecidable (it's actually not even re), use reduction to show the undecidability of**

   $\{ f \mid \forall x\ f(x+1) > f(x) \}$

Sample Question#8

8. **Assume that f and g are both standard Turing Computable (STC). Show that f+g is also STC. You must demonstrate this by writing a new machine in diagrammatic notation. Of course, f and g may be used as submachines.**

Sample Question#9

9. **Let S be a recursive (decidable set), what can we say about the complexity (recursive, re non-recursive, non-re) of T, where $T \subset S$?**

© UCF (Charles E. Hughes)

Sample Question#10

10. Define the pairing function $<x,y>$ and its two inverses $<z>_1$ and $<z>_2$, where if $z = <x,y>$, then $x = <z>_1$ and $y = <z>_2$.

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, $A \le_m B$, if there exists a total recursive function 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 $A \le_m B$ and $B \le_m A$**
- **Sets that are many-one equivalent are in some sense equally hard or easy.**

 © UCF (Charles E. Hughes)

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: $\varnothing$, $\aleph$, all others.
- The hierarchy of undecidable m-1 degrees is an infinite lattice (I'll discuss in class)

One-One Reduction

- **Let A and B be two sets.**
- **We say A one-one reduces to B, $A \leq_1 B$, if there exists a total recursive 1-1 function f such that**
  $x \in A \Leftrightarrow f(x) \in B$
- **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.**

© UCF (Charles E. Hughes)

The Set Halt = K_0 = L_u

- Halt = K_0 = L_u = { <f, x> | [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 $\phi_a(x)\downarrow$ 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 = L_u, is an re (1-1, m-1, Turing) complete set.

© UCF (Charles E. Hughes)

The Set K

- **$K = \{ f \mid \phi_f(f)$ is defined $\}$**
- **Define $f_x(y) = \phi_f(x)$. That is, $f_x(y) = \phi_f(x)$. The index for f_x can be computed from f and x using $S_{1,1}$, where we add a dummy argument, y, to ϕ_f. Let that index be f_x. (Yeah, that's overloading.)**
- **$\langle f,x \rangle \in K_0$ iff $x \in dom(\phi_f)$ iff $\forall y[\phi_{f_x}(y)\downarrow]$ iff $f_x \in K$.**
- **The above provides a 1-1 function that reduces K_0 to K.**
- **Since K_0 is an re (1-1, m-1, Turing) complete set and K is re, then K is also re (1-1, m-1, Turing) complete.**

Reduction and Rice's

Two Interesting Sets

- The sets

  L_r = { x | dom [x] is recursive }

  L_{nr} = { x | dom [x] is not recursive }
- L_r is very easily confused with the set of indices of algorithms. It includes the indices of all algorithms, since their domains (all natural numbers) are clearly recursive. It also includes many indices of functions which diverge at some points where a corresponding algorithm might have produced a 0 output (rejection).
- Our claim is that neither of these sets is re.

L_r is Non-RE

Let HALT(x,y) = ∃t STP(y, x, t)

Consider again the set

L_r = { x | dom [x] is recursive }

Suppose L_r is re. We can show that this implies that the complement of L_u is also re, but then since L_u is re, we would have that L_u is recursive (decidable), an impossibility. We attack this by defining, for each function index x and input y, a function

$F_{x,y}(z) = HALT(x, y) + HALT(\langle z\rangle_1 , \langle z\rangle_2)$

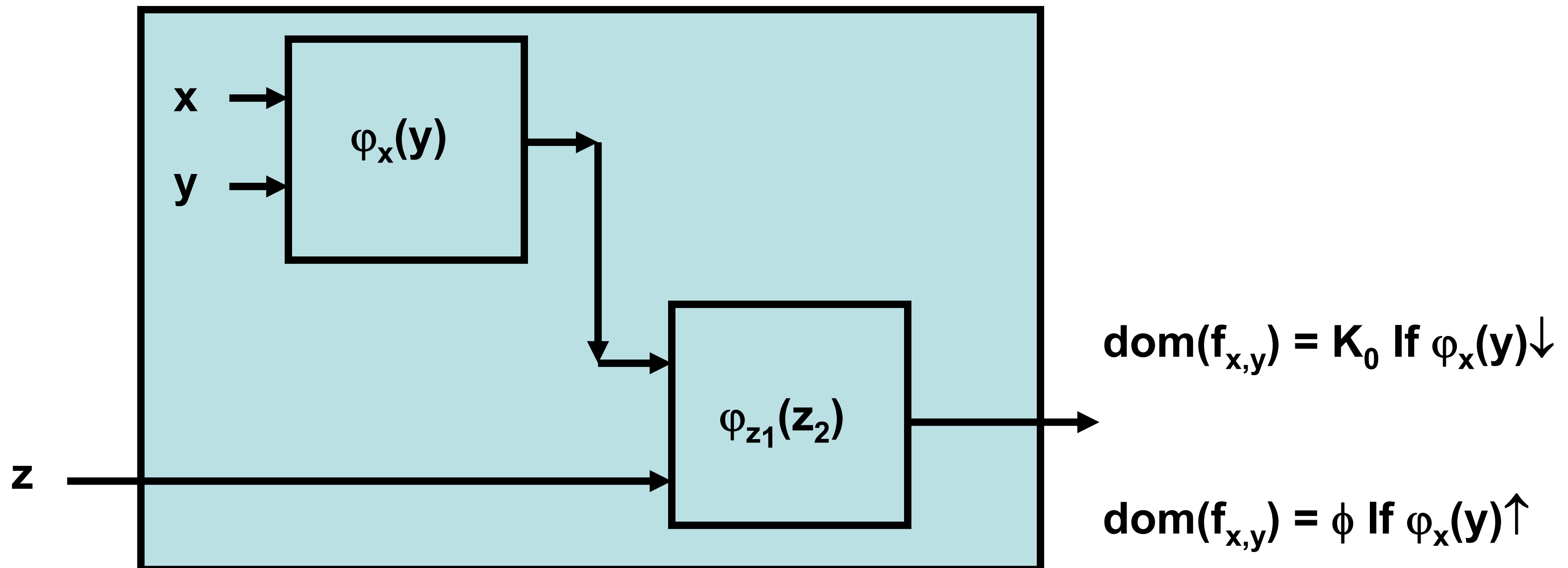
This function's domain is L_u, if [x] (y) is defined, and is Ø, otherwise.

Thus, $F_{x,y}$ accepts a recursive language just in case (x, y) $\notin L_u$ (that is, [x] (y) is undefined). But (x, y) $\notin L_u$ just in case $F_{x,y}$'s index is in L_r.

Thus, a semi-decision procedure for L_r implies one for the complement of L_u. So L_r is not re.

L_r Picture Proof

Given arbitrary **x**, **y**, define the function $\mathbf{f_{x,y}(z) = \varphi_x(y) + \varphi_{z1}(z_2)}$.
The following illustrates $\mathbf{f_{x,y,}}$.
Here, $\mathbf{dom(f_{x,y,}) = \phi}$ **if** $\mathbf{\varphi_x(y)}\uparrow$; = $\mathbf{K_0}$ **if** $\mathbf{\varphi_x(y)}\downarrow$
Thus, $\varphi_x(y)\uparrow$ iff $\mathbf{f_{x,y}}$ is in $\mathbf{L_r}$, and so $\mathbf{\sim K_0} \le_1 \mathbf{L_r}$. If $\mathbf{L_r}$ is re then so is $\mathbf{\sim K_0}$ and hence $\mathbf{K_0}$ and its complement are both re, implying $\mathbf{K_0}$ is recursive, but that cannot be so, Hence $\mathbf{L_r}$ is not re.



© UCF (Charles E. Hughes)

L_{nr} is Non-re

- A similar proof exists to show that L_{nr} is not re. In this case we want a function whose domain is L_u, if [x] (y) is undefined, and is $\aleph$, otherwise.
- I'd like you to think about this one -- not an assignment, rather a challenge. You might consider starting with a function

$$G_{x,y}(z) = \begin{cases} \text{HALT}(x, y) * (z // 2) & \text{if } z \text{ is odd} \\ \text{HALT}(\langle z // 2\rangle_1, \langle z // 2\rangle_2) * (z // 2) & \text{if } z \text{ is even} \end{cases}$$

- But this function's range is L_u, if [x] (y) is undefined, and is $\aleph$, otherwise. That's not quite what we were after -- we need domain, not range -- but let's assume it's on the right track and that we have $F_{x,y}$.
- Thus, $F_{x,y}$ accepts a recursive language just in case (x, y) is in L_u (that is, [x] (y) is defined). But then (x, y) is not in L_u just in case $F_{x,y}$'s index is in L_{nr}.
- Thus, a semi-decision procedure for L_{nr} implies one for the complement of L_u. So L_{nr} is not re.

Either Trivial or Undecidable

- The previous proof shows that we cannot decide if a (partially) recursive function accepts a recursive set. We cannot even decide if it accepts the empty set.
- In general, there's really nothing that we can decide about recursive functions, based purely on their input/output behavior.
- This generalization of what was just done is Rice's Theorem for recursive index sets.
- Let P be some set of re languages, e.g. P = { L | L is infinite re }. We call P a property of re languages since it divides the class of all re languages into two subsets, those having property P and those not having property P. P is said to be trivial if it is empty (this is not the same as saying P contains the empty set) or contains all re languages. Trivial properties are not very discriminating in the way they divide up the re languages (all or nothing).

Rice's Theorem

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

L_P = { x | dom [x] is in P (has property P) }

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

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

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

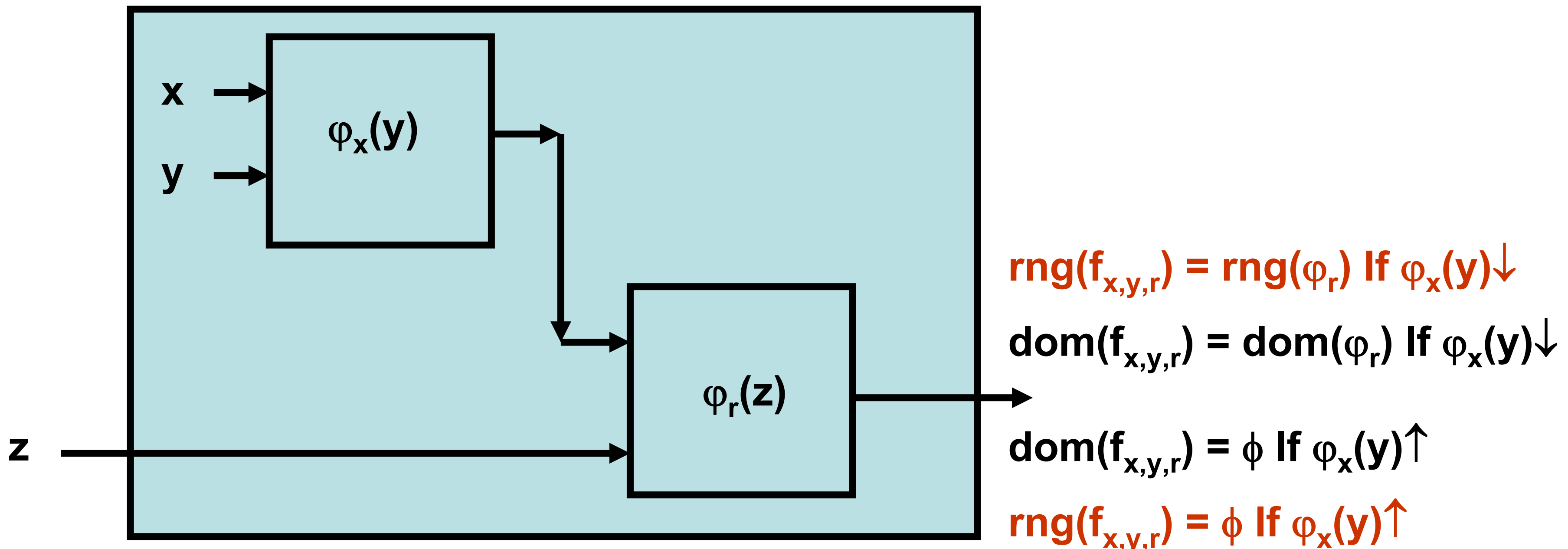
The domain of this function is L if [x](y) converges, otherwise it's Ø. Now if we can determine membership in L_P , we can use this algorithm to decide L_u merely by applying it to $F_{r,x,y}$. An answer as to whether or not $F_{r,x,y}$ has property P is also the correct answer as to whether or not [x](y) 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 $\mathbf{S}_{\mathscr{P}} = \{\mathbf{f} \mid \varphi_\mathbf{f}$ has property $\mathscr{P}\}$, define the function $\mathbf{f}_{x,y,r}(\mathbf{z}) = \varphi_x(\mathbf{y}) - \varphi_x(\mathbf{y}) + \varphi_r(\mathbf{z})$. The following illustrates $\mathbf{f}_{x,y,r}$.
Here, $\mathbf{dom}(\mathbf{f}_{x,y,r}) = \mathbf{dom}(\varphi_r)$ $(\mathbf{f}_{x,y,r}(\mathbf{z}) = \varphi_r(\mathbf{z}))$ **if** $\varphi_x(\mathbf{y})\downarrow$; $= \phi$ **if** $\varphi_x(\mathbf{y})\uparrow$.
Thus, $\varphi_x(\mathbf{y})\downarrow$ iff $\mathbf{f}_{x,y,r}$ has property $\mathscr{P}$, and so $\mathbf{K}_0 \leq_1 \mathbf{S}_{\mathscr{P}}$.



© UCF (Charles E. Hughes)

Corollaries to Rice's

Corollary: The following properties of re sets are undecidable

a) $L = \emptyset$

b) L is finite

c) L is a regular set

d) L is a context-free set

Assignment # 5

1. Let INF = { f | domain(f) is infinite } and NE = { f | there is a y such that f(y) converges}. Show that NE <=m INF. Present the mapping and then explain why it works as desired. To do this, define a total recursive function g, such that index f is in NE iff g(f) is in INF. Be sure to address both cases (f in & f not in)
2. Is INF <=m NE? If you say yes, show it. If you say no, give a convincing argument that INF is more complex that NE.
3. What, if anything, does Rice's Theorem have to say about the following? In each case explain by either showing that all of Rice's conditions are met or convincingly that at least one is not met.
   a.) RANGE = { f | there is a g [range(g) = domain(f)] }
   b.) PRIMITIVE = { f | f's description uses no unbounded mu operations }
   c.) FINITE = { f | domain(f) is finite }

Due: October 22

Canonical Processes, Groups and Grammars

Post Canonical Systems of Varying Sorts and Their Relation to Groups and Grammars

Semi-Groups, Monoids, Groups

S = (G, •) is a semi-group if

G is a set, • is a binary operator, and

1. Closure: If $x,y \in G$ then $x \bullet y \in G$
2. Associativity: $x \bullet (y \bullet z) = (x \bullet y) \bullet z$

S is a monoid if

3. Identity: $\exists e \in G \; \forall x \in G \; [e \bullet x = x \bullet e = x]$

S is a group if

4. Inverse: $\forall x \in G \; \exists x^{-1} \in G \; [x^{-1} \bullet x = x \bullet x^{-1} = e]$

S is Abelian if • is commutative

Finitely Presented

- If S is a semi-group (monoid, group) defined by a finite set of symbols Σ, called the alphabet or generators, and a finite set of equalities ($\alpha_i = \beta_i$), the reflexive transitive closure of which determines equivalence classes over S, then S is a finitely presented semi-group (monoid, group). Note, the set S is the closure of the generators under the semi-group's operator.
- The word problem for S is the problem to determine of two elements α, β, whether or not $\alpha = \beta$, that is, whether or not they are in the same equivalence class.
- If $\bullet$ is commutative, then S is Abelian.

Finitely Presented Monoids

- Strings over an alphabet (operation is concatenation, identity is string of length zero).
- Natural numbers (use alphabet {1} make + the operator, identity is 0 occurrences of a 1, use shorthand that n represents n adds: 1+1+ … +1). This is actually an Abelian monoid.
- In above cases, we would also need rules for equivalence classes, e.g., we can get the equivalences classes dividing the even and odd numbers by
  1+1 = 0
  The two classes have representatives 0 and 1.

Abelian Monoids

- **Consider a finitely presented Abelian monoid over generators $\Sigma = \{a_1,\ldots,a_n\}$.**
- **Since this is Abelian, we can always organize the letters in a word into a canonical form, $a_1^{k_1},\ldots,a_n^{k_n}$, k_i is the number of times a_i appears.**
- **Thus, each word is a vector $<k_1,.., k_n>$ and each rule is a pair of such vectors.**
- **The above can be recast as a FRS, where each rule is bi-directional (vector values are powers of primes) and there is no order. It can also be recast as a bi-directional vector addition system, VAS, where a rule in a VAS is of the form**

  $<j_1,.., j_n> \rightarrow <k_1,.., k_n>$

  This means add $<k_1-j_1,.., k_n-j_n>$ to a vector $<i_1,.., i_n>$, provided $i_s \geq j_s$, $1 \leq s \leq n$.

Thue Systems

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

Semi-Thue Systems

- **Devised by Emil Post**
- **A one-directional version of Thue systems**
- **S = (Σ, R), where Σ is a finite alphabet and R is a finite set of rules of form** $\alpha_i \to \beta_i$, α_i, $\beta_i \in \Sigma^*$
- **We define** $\Rightarrow^*$ **as the reflexive, transitive closure of** $\Rightarrow$, **where w** $\Rightarrow$ **x iff w=y**α**z and x=y**β**z, where** $\alpha \to \beta$

Word Problems

- **Let S = (Σ, R) be some Thue (Semi-Thue) system, then the word problem for S is the problem to determine of arbitrary words w and x over S, whether or not w $\Leftrightarrow^*$ x (w $\Rightarrow^*$ x)**
- **The Thue system word problem is the problem of determining membership in equivalence classes. This is not true for Semi-Thue systems.**
- **We can always consider just the relation $\Rightarrow^*$ since the symmetric property of $\Leftrightarrow^*$ comes directly from the rules of Thue systems.**

© UCF (Charles E. Hughes)

Post Canonical Systems

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

Examples of Canonical Forms

- **Propositional rules**
  $P, P \supset Q \rightarrow Q$
  $\sim P, P \cup Q \rightarrow Q$
  $P \cap Q \rightarrow P$ **oh, oh $a \cap (b \cap c) \Rightarrow a \cap (b$**
  $P \cap Q \rightarrow Q$
  $(P \cap Q) \cap R \leftrightarrow P \cap (Q \cap R)$
  $(P \cup Q) \cup R \leftrightarrow P \cup (Q \cup R)$
  $\sim(\sim P) \leftrightarrow P$
  $P \cup Q \rightarrow Q \cup P$
  $P \cap Q \rightarrow Q \cap P$
- **Some proofs over $\{a,b,(,),\sim,\supset,\cup,\cap\}$**
  $\{a \cup c, b \supset \sim c, b\} \Rightarrow \{a \cup c, b \supset \sim c, b, \sim c\} \Rightarrow$
  $\{a \cup c, b \supset \sim c, b, \sim c, c \cup a\} \Rightarrow$
  $\{a \cup c, b \supset \sim c, b, \sim c, c \cup a, a\}$ which proves "a"

© UCF (Charles E. Hughes)

Simplified Canonical Forms

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

Examples of Post Systems

- **Alphabet Σ = {a,b,#}. Semi-Thue rules:**
  aba $\rightarrow$ b
  #b# $\rightarrow$ λ
  For above, #a^nba^m# $\Rightarrow^*$ λ iff n=m
- **Alphabet Σ = {0,1,c,#}. Normal rules:**
  0c $\rightarrow$ 1
  1c $\rightarrow$ c0
  #c $\rightarrow$ #1
  0 $\rightarrow$ 0
  1 $\rightarrow$ 1
  # $\rightarrow$ #
  For above, *binary*c# $\Rightarrow^*$ *binary+1*# where *binary* is some binary number.

© UCF (Charles E. Hughes)

Simulating Turing Machines

- **This is done in text and will be done in class. Basically, we need at least one rule for each 4-tuple in the Turing machine's description.**
- **The rules lead from one instantaneous description to another.**
- **The Turing ID αqaβ is represented by the string hαqaβh, a being the scanned symbol.**
- **The tuple q a b s leads to qa $\rightarrow$ sb**
- **Moving right and left can be harder due to blanks.**

© UCF (Charles E. Hughes)

Details of Halt(TM) $\leq$ Word(ST)

- **Let M = (Q, {0,1}, T), T is Turing table.**
- **If qabs $\in$ T, add rule qa $\rightarrow$ sb**
- **If qaRs $\in$ T, add rules**
  - qab $\rightarrow$ asb if a$\neq$0 $\forall$b$\in${0,1}
  - qah $\rightarrow$ as0h if a$\neq$0
  - cqab $\rightarrow$ casb if a=0 $\forall$b,c$\in${0,1}
  - hqab $\rightarrow$ hsb if a=0 $\forall$b$\in${0,1}
  - cqah $\rightarrow$ cas0h if a=0 $\forall$c$\in${0,1}
  - hqah $\rightarrow$ hs0h if a=0
- **If qaLs $\in$ T, add rules**
  - bqac $\rightarrow$ sbac $\forall$a,b,c$\in${0,1}
  - hqac $\rightarrow$ hs0ac if $\forall$a,c$\in${0,1}
  - bqah $\rightarrow$ sbah if a$\neq$0 $\forall$c$\in${0,1}
  - bqah $\rightarrow$ sbh if a=0 $\forall$b$\in${0,1}
  - hqah $\rightarrow$ hs0ah if a$\neq$0
  - hqah $\rightarrow$ hs0h if a=0

Semi-Thue Word Problem

- **Construction from TM, M, gets:**
- $h1^xq_10h \Rightarrow_{\Sigma(M)}^* hq_0h$ **iff** $x \in \mathcal{L}(M)$.
- $hq_0h \Rightarrow_{\Pi(M)}^* h1^xq_10h$ **iff** $x \in \mathcal{L}(M)$.
- $hq_0h \Leftrightarrow_{\Sigma(M)}^* h1^xq_10h$ **iff** $x \in \mathcal{L}(M)$.
- **Can recast both Semi-Thue and Thue Systems to ones over alphabet {a,b} or {0,1}**

Assignment # 6

1. **Using reduction from the complement of the Halting Problem, show the undecidability of the problem to determine if an arbitrary partial recursive function, f, has a summation upper bound. This means that there is a M, such that the sum of all values in the range of f (repeats are added in and divergence just adds 0) is ≤ M.**
2. **Use one of the versions of Rice's Theorem to show the undecidability of the problem to determine if an arbitrary partial recursive function, f, has a summation upper bound. This means that there is a M, such that the sum of all values in the range of f (repeats are added in and divergence just adds 0) is ≤ M.**
3. **Show that given a Semi-Thue, S, you can produce a Post Normal System, N_S, such that $x \Rightarrow_S^* y$ iff $\$x \Rightarrow_{N_S}^* \$y$. You must give the construction of N_S from S and a justification of why this meets the condition stated above.**

Due: October 29

Formal Language Review

Pretty Basic Stuff

Closure Properties

- **Regular (Finite State) Languages**
  - Union, intersection, complement, substitution, quotient (with anything), max, min, cycle, reversal
  - Use of Pumping Lemma and Myhill-Nerode
- **Context Free**
  - Union, intersection with regular, substitution, quotient with regular, cycle, reversal
  - Use of Pumping and Ogden's Lemma
- **Context Sensitive Languages**
  - Union, intersection, complement, Epsilon-free substitution, cycle, reversal

© UCF (Charles E. Hughes)

Non-Closure

- **CFLs not closed under**
  - Intersection, complement, max, min

- **CSLs not closed under**
  - Homomorphism (or substitution with empty string), max (similar to homomorphism)

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).**

Formal Language

Undecidability Continued

PCP and Traces

Post Correspondence Problem

- **Many problems related to grammars can be shown to be no more complex than the Post Correspondence Problem (PCP).**
- **Each instance of PCP is denoted: Given n>0, Σ a finite alphabet, and two n-tuples of words**
  $(x_1, \ldots , x_n), (y_1, \ldots , y_n)$ over Σ,
  does there exist a sequence $i_1, \ldots , i_k$, $k>0$, $1 \le i_j \le n$, such that
  $x_{i_1} \ldots x_{i_k} = y_{i_1} \ldots y_{i_k}$?
- **Example of PCP:**
  n = 3, Σ = { a , b }, (a b a , b b , a), (b a b , b , b a a).
  Solution 2 , 3, 1 , 2
  b b a a b a b b = b b a a b a b b

PCP Example#2

- **Start with Semi-Thue System**
  - aba $\rightarrow$ ab; a $\rightarrow$ aa; b $\rightarrow$ a
  - Instance of word problem: bbbb $\Rightarrow$*? aa
- **Convert to PCP**

–	[bbbb*	ab	$\underline{ab}$	aa	$\underline{aa}$	a	$\underline{a}$	]
	[	$\underline{aba}$	aba	$\underline{a}$	a	$\underline{b}$	b	$\underline{*}$aa]
– And		*	$\underline{*}$	a	$\underline{a}$	b	$\underline{b}$	
		$\underline{*}$	*	$\underline{a}$	a	$\underline{b}$	b	

How PCP Construction Works?

- **Using underscored letters (~ in text) avoids solutions that don't relate to word problem instance. E.g.,**

  aba a

  ab   aa

- **Top row insures start with [W_0*****
- **Bottom row insures end with *W_f]**
- **Bottom row matches W_i, while top matches W_{i+1} (one is underscored)**

© UCF (Charles E. Hughes)

Ambiguity of CFG

- **Problem to determine if an arbitrary CFG is ambiguous**

$S \rightarrow A \mid B$

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

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

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

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

- **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. $L_A \cap L_B \neq Ø$, if and only there is a solution to this PCP instance.**

CSG Produces Something

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.

Assignment # 7

1. **Present the description of a PDA (in words) that accepts L_A (see page 253). You may assume that [i] is a single symbol.**
2. **Present the description of a PDA (in words) that accepts $\sim L_A$ (see page 253).**
3. **Use (2) to show that it is undecidable to determine of an arbitrary CFL, L, whether or not $L = \Sigma^*$.**
4. **Prove that Post Correspondence Systems over {a} are decidable.**

Due: November 14

Traces (Valid Computations)

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

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

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

Traces of FRS

- I have chosen, once again to use the Factor Replacement Systems, but this time, Factor Systems with Residues. The rules are unordered and each is of the form
  $a\,x + b \rightarrow c\,x + d$
- These systems need to overcome the lack of ordering when simulating Register Machines. This is done by

j.	$INC_r[i]$	$p_{n+j}\ x$	$\rightarrow p_{n+i}\ p_r\ x$
j.	$DEC_r[s, f]$	$p_{n+j}\ p_r\ x$	$\rightarrow p_{n+s}\ x$
		$p_{n+j}\ pr\ x + k\ p_{n+j}$	$\rightarrow p_{n+f}\ p_r\ x + k\ p_{n+f}$, $1 \leq k < p_r$

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

  $$p_{n+m+1}\ x \rightarrow 0$$

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

  $$0 \rightarrow 0$$

© UCF (Charles E. Hughes)

Intersection of CFLs

- Let (n, ((a1,b1,c1,d1) , … ,(ak,bk,ck,dk)) be some factor replacement system with residues. Define grammars G1 and G2 by using the 4k+2 rules

  G : $F_i \rightarrow 1^{ai}F_i1^{ci}$ | $1^{ai+bi}\#1^{ci+di}$ $1 \le i \le k$

  $S_1 \rightarrow \# F_i S_1$ | $\# F_i \#$ $1 \le i \le k$

  $S_2 \rightarrow \# 1^{x0}S_11^{z0}\#$ Z_0 is 0 for us

  G1 starts with S_1 and G2 with S_2
- Thus, using the notation of writing Y in place of 1^Y,

  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.

Intersection Continued

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

$L1 \cap L2 = \{\#X_0 \# X_1 \# X_2 \# X_3 \# X_4 \# \ldots \# X_{2k-1} \# X_{2k} \# Z_0 \# \}$

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

Quotients of CFLs

- **Let (n, ((a1,b1,c1,d1) , … ,(ak,bk,ck,dk)) be some factor replacement system with residues. Define grammars G1 and G2 by using the 4k+4 rules**

  G : $F_i \rightarrow 1^{ai}F_i1^{ci} \mid 1^{ai+bi}\#1^{ci+di}$ $\quad 1 \le i \le k$

  $T_1 \rightarrow \# F_i T_1 \mid \# F_i \#$ $\quad 1 \le i \le k$

  $A \rightarrow 1 A 1 \mid \$ \#$

  $S_1 \rightarrow \$T_1$

  $S_2 \rightarrow A T_1 \# 1^{z0} \#$ $\quad Z_0$ is 0 for us

  G1 starts with S_1 and G2 with S_2

- **Thus, using the notation of writing Y in place of 1^Y,**

  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) = { $X \$ \#X_0 \# X_1 \# X_2 \# X_3 \# X_4 \# \ldots \# X_{2k-1} \# X_{2k} \# Z_0 \#$ }

  where $X_{2i-1} \Rightarrow X_{2i}$, $1 \le i \le k$ and $X = X_0$

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

Finish Quotient

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

L2 / L1 = { X | the system F halts on zero }.

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

© UCF (Charles E. Hughes)

Traces and Type 0

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

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

CSG and Undecidability

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

S'	$\rightarrow$	e S e	
D x	$\rightarrow$	x D	when x $\in$ V $\cup$ T
D e	$\rightarrow$	e d	push the delete characters to far right
α	$\rightarrow$	β	where $\alpha \rightarrow \beta \in$ P and $\lvert\alpha\rvert \leq \lvert\beta\rvert$
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.**

© UCF (Charles E. Hughes)

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 =∅, for CSL, L?
- Is L=Σ*, 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?

Word to Grammar Problem

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

  G: $S \rightarrow h1^x q_1 0h$

  $hq_0h \rightarrow V$

  $V \rightarrow aV$

  $V \rightarrow \lambda$

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

Consequences for Grammar

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

© UCF (Charles E. Hughes)

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

 © UCF (Charles E. Hughes)

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 = Σ*?

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

Undecidability of Finite Convergence for Operators on Formal Languages

Relation to Real-Time (Constant Time) Execution

Simple Operators

- **Concatenation**
  - $A \bullet B = \{ xy \mid x \in A \ \& \ y \in B \}$
- **Insertion**
  - $A \triangleright B = \{ xyz \mid y \in A, xz \in B, x, y, z \in \Sigma^* \}$
  - Clearly, since x can be λ, $A \bullet B \subseteq A \triangleright B$

K-insertion

- $A \triangleright^{[k]} B = \{ x_1y_1x_2y_2 \dots x_ky_kx_{k+1} \mid y_1y_2 \dots y_k \in A, x_1x_2 \dots x_kx_{k+1} \in B, x_i, y_j \in \Sigma^* \}$
- Clearly, $A \bullet B \subseteq A \triangleright^{[k]} B$, for all k>0

Iterated Insertion

- **$A\ (1) \triangleright^{[n]}\ B = A \triangleright^{[n]}\ B$**
- **$A\ (k+1) \triangleright^{[n]}\ B = A \triangleright^{[n]}\ (A\ (k) \triangleright^{[n]}\ B)$**

Shuffle

- **Shuffle (product and bounded product)**
  - $A \Diamond B = \cup_{j \geq 1} A \triangleright^{[j]} B$
  - $A \Diamond^{[k]} B = \cup_{1 \leq j \leq k} A \triangleright^{[j]} B = A \triangleright^{[k]} B$

- **One is tempted to define shuffle product as $A \Diamond B = A \triangleright^{[k]} B$ where**
  $k = \mu\, y\, [\, A \triangleright^{[j]} B = A \triangleright^{[j+1]} B\,]$
  but such a k may not exist – in fact, we will show the undecidability of determining whether or not k exists

More Shuffles

- **Iterated shuffle**
  - $A \diamond^{0} B = A$
  - $A \diamond^{k+1} B = (A \diamond^{[k]} B) \diamond B$

- **Shuffle closure**
  - $A \diamond^{*} B = \cup_{k \geq 0} (A \diamond^{[k]} B)$

Crossover

- **Unconstrained crossover is defined by**
  $A \otimes_u B = \{ wz, yx \mid wx \in A \text{ and } yz \in B\}$

- **Constrained crossover is defined by**
  $A \otimes_c B = \{ wz, yx \mid wx \in A \text{ and } yz \in B, |w| = |y|, |x| = |z| \}$

Who Cares?

- People with no real life (me?)
- Insertion and a related deletion operation are used in biomolecular computing and dynamical systems
- Shuffle is used in analyzing concurrency as the arbitrary interleaving of parallel events
- Crossover is used in genetic algorithms

Some Known Results

- **Regular languages, A and B**
  - A • B is regular
  - A $\triangleright^{[k]}$ B is regular, for all k>0
  - A ◇ B is regular
  - A ◇* B is not necessarily regular
    - Deciding whether or not A ◇* B is regular is an open problem

© UCF (Charles E. Hughes)

More Known Stuff

- **CFLs, A and B**
  - A • B is a CFL
  - A ▷ B is a CFL
  - A $\triangleright^{[k]}$ B is not necessarily a CFL, for k>1
    - Consider A=$a^n b^n$; B = $c^m d^m$ and k=2
    - Trick is to consider (A $\triangleright^{[2]}$ B) ∩ a*c*b*d*
  - A ◇ B is not necessarily a CFL
  - A ◇* B is not necessarily a CFL
    - Deciding whether or not A ◇* B is a CFL is an open problem

Immediate Convergence

- $L = L^2$?
- $L = L \triangleright L$?
- $L = L \diamond L$?
- $L = L \diamond^* L$?
- $L = L \otimes_c L$?
- $L = L \otimes_u L$?

© UCF (Charles E. Hughes)

Finite Convergence

- $\exists k>0\ L^k = L^{k+1}$
- $\exists k\geq 0\ L\ (k) \triangleright L = L\ (k+1) \triangleright L$
- $\exists k\geq 0\ L \triangleright^{[k]} L = L \triangleright^{[k+1]} L$
- $\exists k\geq 0\ L \diamond^{k} L = L \diamond^{k+1} L$
- $\exists k\geq 0\ L\ (k) \otimes_c L = L\ (k+1) \otimes_c L$
- $\exists k\geq 0\ L\ (k) \otimes_u L = L\ (k+1) \otimes_u L$

- $\exists k\geq 0\ A\ (k) \triangleright B = A\ (k+1) \triangleright B$
- $\exists k\geq 0\ A \triangleright^{[k]} B = A \triangleright^{[k+1]} B$
- $\exists k\geq 0\ A \diamond^{k} B = A \diamond^{k+1} B$
- $\exists k\geq 0\ A\ (k) \otimes_c B = A\ (k+1) \otimes_c B$
- $\exists k\geq 0\ A\ (k) \otimes_u B = A\ (k+1) \otimes_u L$

Finite Power of CFG

- **Let G be a context free grammar.**
- **Consider $L(G)^n$**
- **Question1: Is $L(G) = L(G)^2$?**
- **Question2: Is $L(G)^n = L(G)^{n+1}$, for some finite $n>0$?**
- **These questions are both undecidable.**
- **Think about why question1 is as hard as whether or not L(G) is Σ^*.**
- **Question2 requires much more thought.**

© UCF (Charles E. Hughes)

1981 Results

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

Proof #1

- **Question: Does L • L get us anything new?**
  - i.e., Is L • L = L?
- **Membership in a CSL is decidable.**
- **Claim is that $L = \Sigma^*$ iff**

  (1) $\Sigma \cup \{\lambda\} \subseteq L$; and

  (2) L • L = L
- **Clearly, if $L = \Sigma^*$ then (1) and (2) trivially hold.**
- **Conversely, we have $\Sigma^* \subseteq L^* = \cup_{n\geq 0} L^n \subseteq L$**
  - first inclusion follows from (1); second from (2)

© UCF (Charles E. Hughes)

Subsuming •

- **Let ⊕ be any operation that subsumes concatenation, that is $A \bullet B \subseteq A \oplus B$.**
- **Simple insertion is such an operation, since $A \bullet B \subseteq A \triangleright B$.**
- **Unconstrained crossover also subsumes •,**
  $A \otimes_c B = \{ wz, yx \mid wx \in A \text{ and } yz \in B\}$

© UCF (Charles E. Hughes)

$L = L \oplus L$?

- **Theorem 2:**
  The problem to determine if $L = \Sigma^*$ is Turing reducible to the problem to decide if $L \oplus L \subseteq L$, so long as $L \bullet L \subseteq L \oplus L$ and L is selected from a class of languages C over Σ for which we can decide if $\Sigma \cup \{\lambda\} \subseteq L$.

© UCF (Charles E. Hughes)

Proof #2

- **Question: Does $L \oplus L$ get us anything new?**
  - i.e., Is $L \oplus L = L$?
- **Membership in a CSL is decidable.**
- **Claim is that $L = \Sigma^*$ iff**

  (1) $\Sigma \cup \{\lambda\} \subseteq L$; and

  (2) $L \oplus L = L$
- **Clearly, if $L = \Sigma^*$ then (1) and (2) trivially hold.**
- **Conversely, we have $\Sigma^* \subseteq L^* = \cup_{n\geq 0} L^n \subseteq L$**
  - first inclusion follows from (1); second from (1), (2) and the fact that $L \bullet L \subseteq L \oplus L$

Exam#2 Review

Material

- You are responsible for material that was covered on Exam#1 and the next few days (reducibility and Rice's Theorem).
- Study notes through page 227.
- Look back at old exam. This one will be similar, except that it will include some questions as you'll see on the next few pages.

Sample Question#1

1. **Assume A $\leq_m$ B and B $\leq_m$ C. Prove A $\leq_m$ C.**

Sample Question#2

2. **Let Incr = { f | $\forall$x, $\phi_f(x+1)>\phi_f(x)$ }.**
   Let TOT = { f | $\forall$x, $\phi_f(x)\downarrow$ }.
   Prove that Incr $\equiv_m$ TOT.

© UCF (Charles E. Hughes)

Sample Question#3

3. **Let Incr = { f | $\forall x\ \phi_f(x+1)>\phi_f(x)$ }. Use Rice's theorem to show Incr is not recursive.**

Sample Question#4

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

Sample Exam#2 Q1

1. **Choosing from among (REC) recursive, (RE) re non-recursive, (CO) co-re non-recursive, (NR) non-re, categorize each of the sets in a) through d). Justify your answer by showing some minimal quantification of some known recursive predicate or by another clear and convincing short argument.**

a.) { f | domain(f) is infinite } ***NR***

Justification: $\forall x\ \exists <y, t>\ [\ STP(y, f, t)\ \&\&\ y>x\ ]$

b.) { f | f converges in 10 steps for some input x } ***RE***

Justification: $\exists x\ [\ STP(x, f, 10)\ ]$

c.) { f | f converges in 10 steps for some input x<10 } ***REC***

Justification: $\exists x< 10\ [\ STP(x, f, 10)\ ]$

d.) { f | domain(f) is empty } ***CO***

Justification: $\forall <x, t>\ [\ \sim STP(x, f, t)\ ]$

© UCF (Charles E. Hughes)

Sample Exam#2 Q2

2. Let set A and B be each re non-recursive. Consider $C = A \cap B$. For each part, either show sets A and B with the specified property or present a demonstration that this property cannot hold.

a.) Can C be recursive? ***YES***
$A = \{ 2x \mid x \in K\}$; $B = \{2x+1 \mid x \in K\}$ are each 1-1 equivalent to K and so are re, non-recursive.
$C = A \cap B = \varnothing$, which is clearly recursive.

b.) Can C be re non-recursive? ***YES***
$A = K$; $B = K$; $C = A \cap B = K$ which is re, non-recursive,

c.) Can C be non-re? ***NO***
Let f_A semi-decide A; f_B semi-decide B.
That is, $x \in A \Leftrightarrow f_A(x)\downarrow$ and $x \in B \Leftrightarrow f_B(x)\downarrow$

Define $f_C(x) = f_A(x) + f_B(x)$
$f_C(x)\downarrow \Leftrightarrow (f_A(x) + f_B(x))\downarrow \Leftrightarrow f_A(x)\downarrow$ && $f_B(x)\downarrow \Leftrightarrow x \in A$ && $x \in B \Leftrightarrow x \in A \cap B \Leftrightarrow x \in C$

Thus, f_C is a semi-decision procedure for C, proving that C must be re.

Sample Exam#2 Q3

3. **Let set A and B be sets, such that A $\leq_m$ B. Answer the following, justifying your answers.**

   Assume A is non-recursive. What does this say about the complexity of B?
   B is non-recursive. Assume otherwise.
   Since A $\leq$m B then $\exists$ total recursive function f | $\forall$x x $\in$ A $\Leftrightarrow$ f(x) $\in$ B.
   If B were recursive and had a characteristic function (algorithm) χ_B then we could solve A by χ_A(x) =χ_B(f(x)), but that contradicts B being non-recursive.)

   Assume B is non-recursive. What does this say about the complexity of A?
   This says nothing about A's non-recursiveness. It does say that A is no worse that B, but we haven't even bounded B's complexity to be re. As an example, if A = $\aleph$ and b $\in$ B (B must be non-empty) then A $\leq_m$ B by f(x) = b, $\forall$x. Of course, A could be non-recursive. For example, if A=B, then A $\leq_m$ B by f(x) = x, $\forall$x

Sample Exam#2 Q4

4. **Assume S is the range of some partial recursive function f_S. Prove that S is the domain <u>and</u> range of some partial recursive function g_S. To get full credit, you must argue convincingly (not formally) that the function you specified is the correct one for S. You may use common known recursive functions to attack this (e.g., STP, VALUE, UNIV), but you may not use known equivalent definitions of enumerable or semi-decidable.**

Define $g_S(x) = (\exists <y,t> [\ STP(y, f_S, t)\ \&\&\ Value(y, f_S, t) == x\ ]\ ) * x$

$g_S(x)$ *either diverges or equals x.*

$g_S(x) = x \Leftrightarrow g_S(x)\downarrow$

$\Leftrightarrow \exists <y,t> [\ STP(y, f_S, t)\ \&\&\ Value(y, f_S, t) == x\ ]$

$\Leftrightarrow \exists y\ f_S(y) == x \Leftrightarrow x \in range(f_S)$

Therefore $x \in range(g_S) \Leftrightarrow x \in domain(g_S) \Leftrightarrow x \in domain(f_S)$

Sample Exam#2 Q5

5. Let INFINITE = { f | domain(f) is infinite } and NE = { f | $\exists$y φ_f(y)$\downarrow$ }. Show that NE $\leq_m$ INFINITE. Present the mapping and then explain why it works as desired.

Define $g_f(x) = \mu$ <y,t> STP(y, f, t)

f $\in$ NE $\Rightarrow$ $\exists$ <y,t> STP(y, f, t)

Let k = μ <y,t> STP(y, f, t)

Then $g_f(x)$ = k $\forall$x and g_f $\in$ INFINITE

f $\notin$ NE $\Rightarrow$ $\forall$ <y,t> ~STP(y, f, t) $\Rightarrow$

$\forall$ x $g_f(x)\uparrow$ $\Rightarrow$ g_f $\notin$ INFINITE

Thus, NE $\leq_m$ INFINITE as was required.

Sample Exam#2 Q6a,b

6. **What, if anything, does Rice's Theorem have to say about the following? In each case explain by either showing that all of Rice's conditions are met or convincingly that at least one is not met.**

***a.*) RANGE = { f | $\exists$ g [range(φ_g) = domain(φ_f)] }**

This is trivial since, as shown in course and assignments and first exam, the property holds for all f. The simple thing to do is to define g(x) = f(x) – f(x) + x. This means that Rice's Theorem says nothing about RANGE.

***b.*) PRIMITIVE = { f | f's description uses no unbounded μ operations }**

This is non-trivial – F1(x) = x $\in$ PRIMITIVE but F2(x) = μ y[x == y] $\notin$ PRIMITIVE.

However, PRIMITIVE is not an I/O property. Revisiting the two functions above,

$\forall$ x F1(x) = F2(x) = x, but one is in and the other is out of PRIMITIVE. Thus, this is not an I/O property and Rice's Theorem says nothing about PRIMITIVE.

Sample Exam#2 Q6c

c.) FINITE = { f | domain(f) is finite }

Non-Trivial: $\uparrow(x) = \mu\, y[x == x+1] \in$ FINITE; $s(x) = x+1 \notin$ FINITE.

I/O Property: Let f, g be arbitrary prf's such that $\forall x\ f(x) = g(x)$ (meaning if one converges, both do and produce the same value; but if one diverges, both do).

f $\in$ FINITE	*$\Leftrightarrow$ domain(f) is finite*	*Definition of FINITE*
	$\Leftrightarrow$ domain(g) is finite	*Since domain(f) = domain(g)*
	$\Leftrightarrow$ g $\in$ FINITE	*Definition of FINITE*

Thus, Rice's Theorem applies, proving that FINITE is non-recursive.

Term Rewriting

Types of Rewriting

- String rewriting is just grammars and the variety of rewriting systems posed by Post. In fact. L-systems are a form of concurrent string rewriting.
- Graph rewriting systems are often used in various forms of analyzers and optimizers, e.g., compiler optimizers.
- Some rewriting systems have a knowledge base underlying them, e.g., about operations on numbers. Such systems often are used as programming languages in systems that seek to simplify expressions, e.g., in Mathematica.

 © UCF (Charles E. Hughes)

Term Rewriting Systems

- These have equations as rules, but they are intended to be be rewritten in one direction only (lhs matches subterm which is replaced by rhs).
- Matching is a form of unification (as in theorem proving and Prolog).

TRS (Ackerman's Function)

R1: $f(0,y) \rightarrow y+1$

R2: $f(x+1,0) \rightarrow f(x,1)$

R3: $f(x+1,y+1) \rightarrow f(x,f(x+1,y))$

For all x, y in $\aleph$.

$f(0,y) \Rightarrow y+1$ by R1

$f(1,y) \Rightarrow f(0,f(1,y-1))$ if $y>0$

$\Rightarrow f(1,y-1)+1 \Rightarrow \ldots \Rightarrow f(1,0) + y \Rightarrow f(0,1) + y \Rightarrow y+2$

$f(2,y) \Rightarrow f(1,f(2,y-1))$ if $y>0$

$\Rightarrow \ldots \Rightarrow f(2,y-1)+2 \Rightarrow \ldots \Rightarrow 2y+3$

Thus, $f(2,3) \Rightarrow 9$

Process

- **Each element is a term**
- **Each term is rewritten by an equation**
- **Each application of an equation is based on a substitution, e.g.,**
  $f(1,2)\ [x \rightarrow 0,\ y \rightarrow 1 \text{ in } f(x+1,y+1) \rightarrow f(x,f(x+1,y))]$
  $\Rightarrow f(0,f(1,1))$
- **Equations are only applied left to right**

Sort by Rewriting

max(0,x) $\rightarrow$ x

max(x,0) $\rightarrow$ x

max(s(x),s(y)) $\rightarrow$ s(max(x,y))

min(0,x) $\rightarrow$ x

min(x,0) $\rightarrow$ x

min(s(x),s(y)) $\rightarrow$ s(min(x,y))

sort(λ) $\rightarrow$ λ

sort(x : y) $\rightarrow$ insert(x, sort(y))

insert(x,λ) $\rightarrow$ x : λ

insert(x,y : z) $\rightarrow$ max(x,y) : insert(min(x,y), z)

Example Sort

- **sort(5:2:3:e) →**
- **insert(5,sort(2:3:e)) →**
- **insert(5,insert(2,sort(3:e))) →**
- **insert(5,insert(2,insert(3:e))) →**
- **insert(5,insert(2,3:e)) →**
- **insert(5,max(2,3):insert(min(2,3),e)) → …**
- **insert(5,3:insert(2,e)) →**
- **insert(5,3:2:e))) →**
- **5:insert(3,2:e) → …**
- **5:3:insert(2:e) → …**
- **5:3:2:e**

 © UCF (Charles E. Hughes)

Simplification

0 + X = X

succ(X) + Y = succ(X + Y)

0 * X = 0

succ(X) * Y = X * Y + Y

2 * 3 =

succ(succ(0)) * 3 =
(succ(0)*3) + 3 = 0*3 + 3 + 3 =
0 + 3 + 3 = 3 + 3 =
… = succ(succ(succ(3))) = 6

Assumes knowledge of simple counting by 1.

Differentiation

Consider the rewriting rules

	Rule	Condition
1.	$x + 0 \to x$	
2.	$0 + x \to x$	
3.	$x \times 0 \to 0$	
4.	$0 \times x \to 0$	
5.	$x \times 1 \to x$	
6.	$1 \times x \to x$	
7.	$P(x, 1) \to x$	
8.	$P(x, 0) \to 1$	
9.	$D(n, x) \to 0$	where n is any constant
10.	$D(x, x) \to 1$	
11.	$D(y + z, x) \to D(y, x) + D(z, x)$	
12.	$D(y \times z, x) \to y \times D(z, x) + z \times D(y, x)$	
13.	$D(P(x, n), x) \to n \times P(x, n-1)$	where n is any constant
14.	$D(y, x, k) \to D(D(y, x, k-1), x)$	provided k is a constant and k>1
15.	$D(y, x, 1) \to D(y, x)$	

Rewrite $D(P(x, 2) + P(x, 1), x, 2)$ until it terminates.
Assume normal precedence of arithmetic operators.
This is non-deterministic.

Canonical Systems

- **A terminating TRS is one where all starting terms t lead eventually to a term t' for which no equations apply**
- **A confluent TRs is one where,
if t⇒*t1 and t ⇒*t2 then there is a t' such that t1⇒*t' and t2 ⇒*t'**
- **A terminating, confluent trs is called canonical**
- **Canonical systems are useful in computation because they always halt and always produce a single result**
- **Neither confluence nor termination is decidable, but they are for some restricted systems, e.g., ones where the rhs of all equations have no variables**

Lindenmayer systems

Grammars and Biology
Modeling Plants
Massively inspired by

Prusinkiewicz & Lindenmayer

The algorithmic beauty of plants, 1990, Springer - Verlag

Available online at:

http://algorithmicbotany.org/papers/

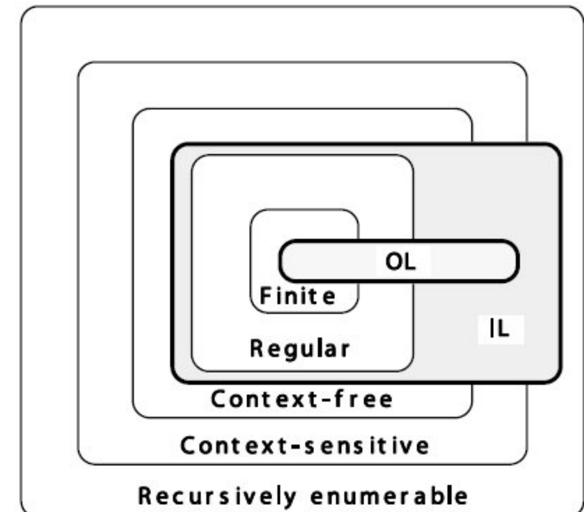
Aristid Lindenmayer (biologist and botanist)

- **Worked on the growth patterns of yeast, filamentous fungi and algae**
- **Formal description of the development of such simple multicellular organisms**
- **Extended to describe complex branching structures and plants**

© UCF (Charles E. Hughes)

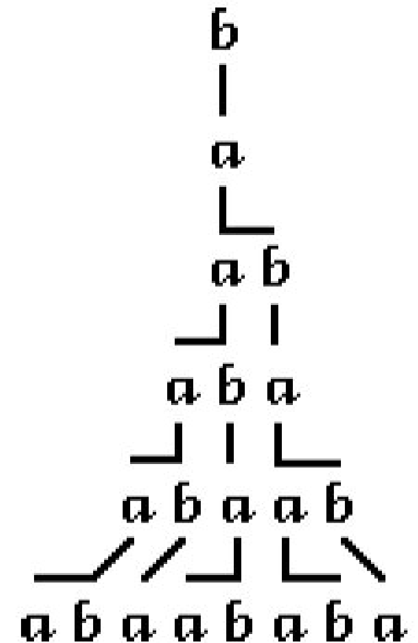
What are L-systems?

- **String-rewriting systems**
- **Parallel application of the rules**
  - Reflects the biological motivation
  - Captures cell divisions occurring at the same time



The first L-system?

- Lindenmayer's original L-system for modeling the growth of algae.
  - variables : A B
  - Axiom ω : B
  - productions : (A → AB), (B → A)
- which produces:
  - n=0 : B → A
  - n=1 : A → AB
  - n=2 : AB → ABA
  - n=3 : ABA → ABAAB
  - n=4 : ABAAB → ABAABABA



Turtle Interpretation

State of the turtle: (x, y, α)

(x, y): *Cartesian position* of the turtle

α: *heading* of the turtle, i.e. the direction in which it is heading

Also

d: *step size*

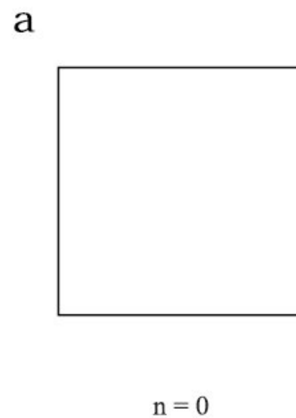
δ: *angle increment*

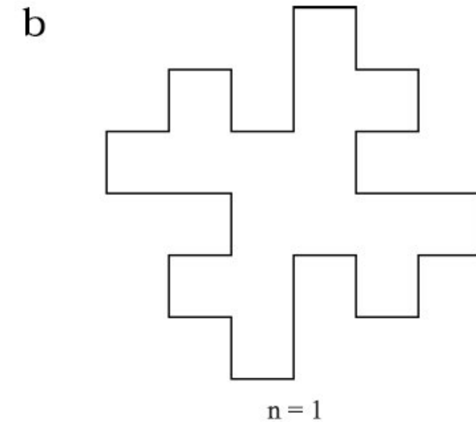
Commands:

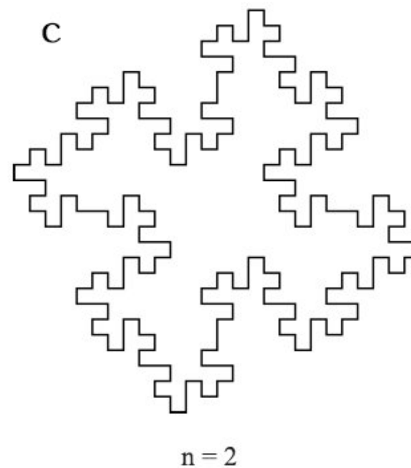
F	move forward a step of length **d** drawing a line segment.
f	the same without drawing.
+	turn left by angle δ.
-	turn right.
[	Push state
]	Pop state

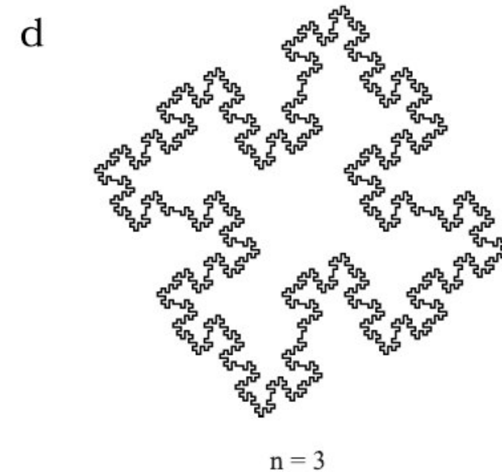
Koch island

- **ω:** F-F-F-F
- **p:**
  F→ F-F+F+FF-F-F+F
- **δ**=90°
- **d** is decreased 4 times between each derivation step









 © UCF (Charles E. Hughes)

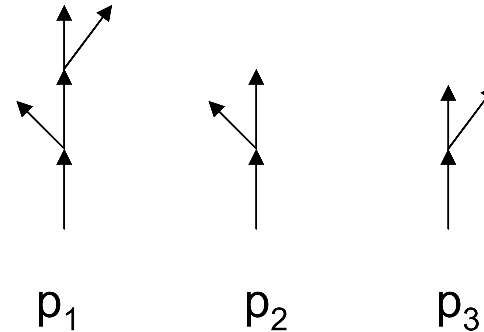
Branching structures

ω: F

p_1: F → F[+F]F[−F]F : .33

p_2: F → F[+F]F : .33

p_3: F → F[−F]F : .34



- ***[*** and ***]*** create a branching structure
- Probabilities of application are Added at the end of the rules
- A single L-system creates a variety of plants

© UCF (Charles E. Hughes)

L-Systems for trees

ω: $FA(1)$

p_1: $A(k) \rightarrow \quad /(\varphi)\ [+(\alpha)\ FA(k+1)]$
$-(\beta)\ FA(k+1)$:
$\min\{1, (2k+1)/k^2\}$

p_2: $A(k) \rightarrow \quad /(\varphi)\ -(\beta)\ FA(k+1)$:
$\max\{0, 1-(2k+1)/k^2\}$

© UCF (Charles E. Hughes)

Interpretation

axiom ω

Module *F* is a branch segment

Module *A*(k) is an apex.

- **This module grows the tree**
- **k is the generation step**

Modules +, – denotes turn

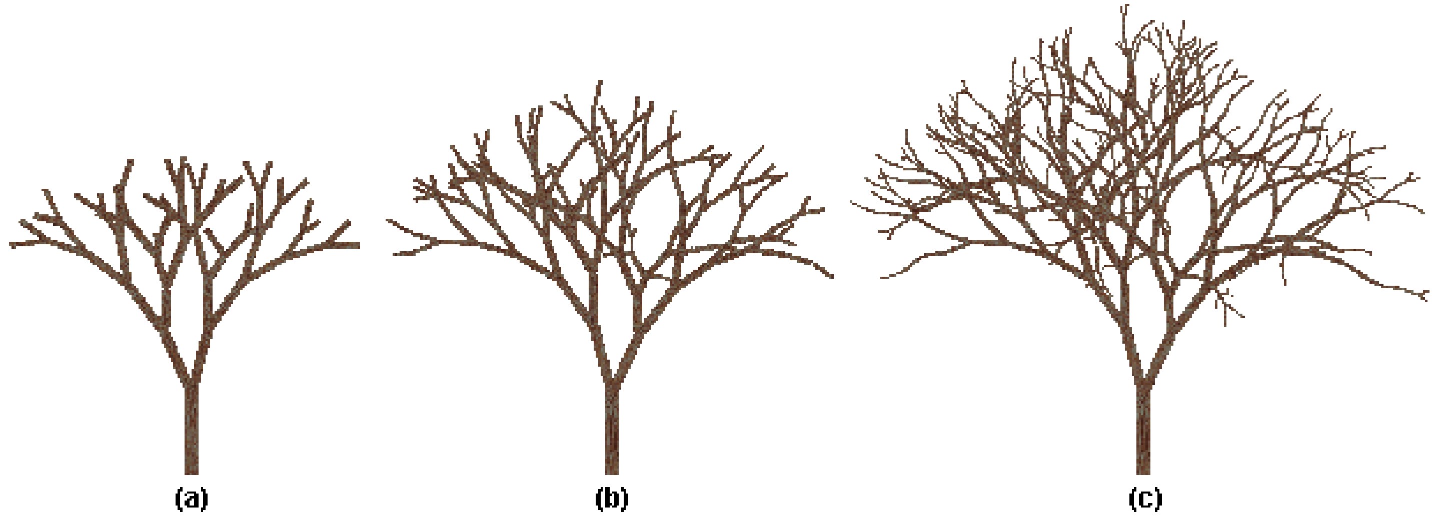
Module / denotes twist

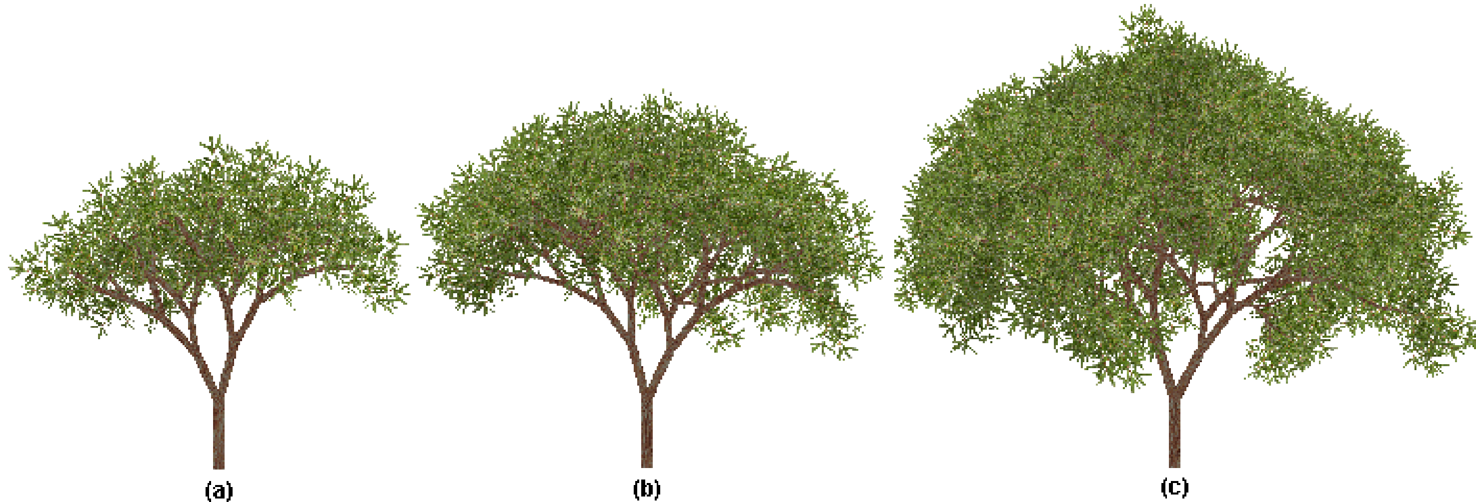
The mean angles for the rotations are specified for a given class of trees (α = 32°, β =20°, φ = 90°).

Module *A*(*k*) can be rewritten non-deterministically

- **p_1 produces 2 branches; $\text{prob}_1 = \min\{1, (2k + 1)/k^2\}$**
- **p_2 produces a single branch segment; probability = $1 - \text{prob}_1$**

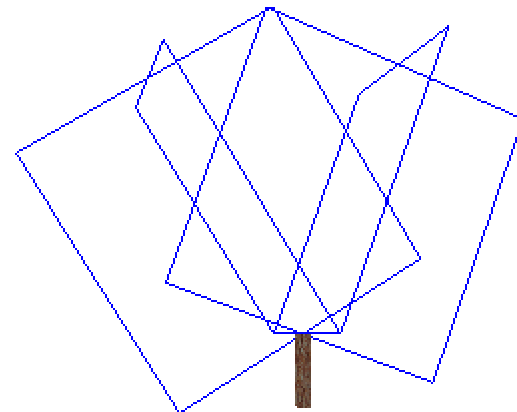
Generations of a Single Tree





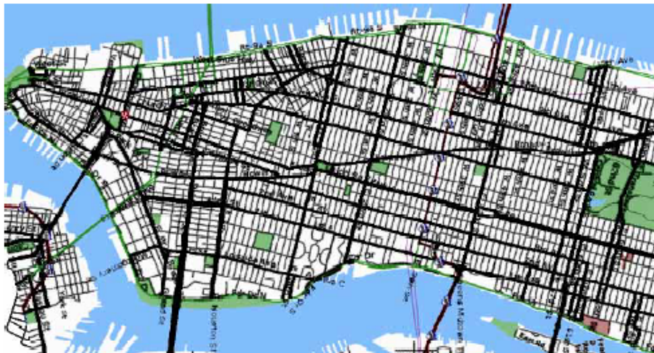
Tree LOD

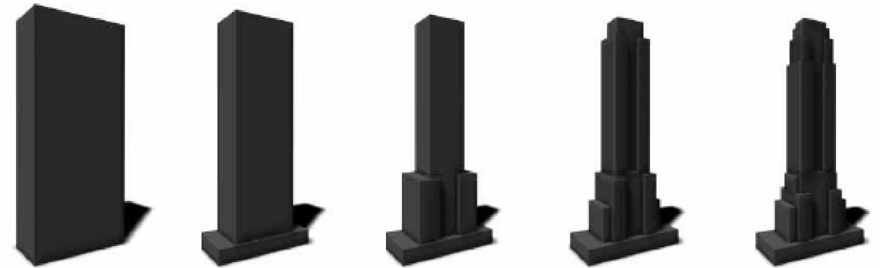
- **Hierarchical**
  - geometry is replaced by productions
    - for example, all geometry due to the symbols introduced in the 10th iteration is replaced
  - geometry is replaced with textured impostors
    - cross polygons

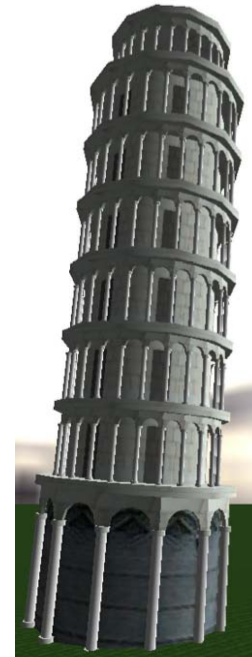


Environment-sensitive

The creation of urban environments







© UCF (Charles E. Hughes)

Bibliography

- **Prusinkiewicz & Lindenmayer**

  The algorithmic beauty of plants, 1990, Springer – Verlag

- **Prusinkiewicz et al.**

  L-systems and beyond, Siggraph 2003 course notes

  Both available online at:

  http://algorithmicbotany.org/papers/

 © UCF (Charles E. Hughes)

Halting vs Mortality

- **The Halting Problem (Set Halt)**
  - Given an arbitrary machine M and starting configuration C, does M halt eventually when started on C
- **The Uniform Halting Problem (Set Total)**
  - Given an arbitrary machine M does M halt eventually no matter what finite configuration it is started on?
- **The Mortality Problem (Set Mortal)**
  - Given an arbitrary machine M does M halt eventually no matter what configuration (finite or infinite) it is started on?

Finite vs Infinite?

- **Consider the machine that computes x+1, given input x, leaving its input unaltered.**
- **Unary notation. Copy x 1's. Append a 1.**
  - On finite input x, machine eventually halts
  - But, given a tape with an infinite number of 1's, this never stops.

© UCF (Charles E. Hughes)

Turing Machine Real-Time Set

- **CTime = RT = { M | ∃K [M halts in at most K steps independent of its starting configuration] }**
- **RT cannot be shown undecidable by Rice's Theorem as it breaks property 2**
  - Choose M1 and M2 to each Standard Turing Compute (STC) ZERO
  - M1 is R (move right to end on a zero)
  - M2 is $\mathcal{L}\ \mathcal{R}$ R (time is dependent on argument)
  - M1 is in RT; M2 is not in RT but they have same I/O behavior, so RT does not adhere to property 2

© UCF (Charles E. Hughes)

Analyzing with Quantifiers

- **CTime = RT = { M | $\exists$K $\forall$C [STP(C, M, K)] }**
- **This would appear to imply that RT is not even re. However, a TM that only runs for K steps can only scan at most K distinct tape symbols. Thus, if we use unary notation, RT can be expressed**
- **CTime = RT = { M | $\exists$K $\forall C_{|C| \le K}$ [STP(C, M, K)] }**
- **We can dovetail over the set of all TMs, M, and all K, listing those M that halt in constant time.**

Immortality Problem

- The immortality problem for Turing machines is the problem to determine of an arbitrary TM, M, if there exist a configuration (not necessarily finite) that causes M to run forever.
- Its complement, the mortality problem is re, non-recursive and this is the basis of our proof.

© UCF (Charles E. Hughes)

Infinite Configurations

- **Consider the Turing machine**
  - $\mathcal{L}\ \mathcal{R}$ R
- **This is just our ZERO machine of a few pages ago.**
- **On a finitely marked tape, this machine is mortal, but on an infinitely marked one it can be immortal.**

© UCF (Charles E. Hughes)

Hooper's Result

- **Theorem 3 (Hooper 1966):**
  Mortal is re undecidable
- **Note, the seemingly related problem of determining if a Turing machine has any finite immortal configurations is the complement of TOT and is not even re.**
- **Unfortunately, Hoopers' proof is quite complex, so we'll just accept the result.**

Mortal and RT

Theorem 4: The set of mortal TMs is exactly the same as the set of TM in RT.

Proof: If M∈RT then M∈MORTAL, so RT ⊂ MORTAL.

Let M∉RT. If any finite ID does not lead to a halt, then M∉MORTAL. Assume then that all finite IDs cause M to halt. Let $\mathcal{D}$ be the set of IDs such that, if M starts on d∈$\mathcal{D}$, it will eventually scan all of d, before scanning any other square of the tape. Let $\{q_1,\ldots,q_m\}$ be the states of M. We define a forest of m trees, one for each state, such that the jth tree has root q_j. If $d_0,d_1 \in \mathcal{D}$ and q_j is a symbol of d_0 and d_1 and $d_1=\sigma d_0$ or $d_1=d_0\sigma$ where σ is a tape symbol, then d_0 is a parent of d_1 in the jth tree.
(Continued)

© UCF (Charles E. Hughes)

Mortal and RT (continued)

Note that when M starts in d_1, the square containing σ is scanned after every other square of d_1, but before any square not in d_1. Since M is not in RT but every finite ID causes it to halt, at least one of the trees of the forest must be infinite. Since the degree of each vertex is finite (bounded by the number of tape symbols), at least one tree must have an infinite branch. Therefore, there exists an infinite ID that causes M to travel an infinite distance on the tape. It follows that M$\notin$MORTAL, and so if M$\notin$RT then M$\notin$MORTAL. Hence, MORTAL $\subset$ RT.

Combining the two parts, RT = MORTAL.

© UCF (Charles E. Hughes)

Consequences

- **We cannot decide if the set of valid terminating traces of an arbitrary machine M are finite.**

- **Put differently, we cannot decide if there is an upper bound on the length of any valid trace.**

© UCF (Charles E. Hughes)

1981, Again

- **Theorem 5**:
  The problem to determine, for an arbitrary context free language L, if there exist a finite n such that $L^n = L^{n+1}$ is undecidable.

© UCF (Charles E. Hughes)

L for Machine M

- $L_1 = \{ C_1\# C_2^R \$ \mid C_1, C_2$ are configurations $\}$,
- $L_2 = \{ C_1\#C_2^R\$C_3\#C_4^R \ \ldots \ \$C_{2k-1}\#C_{2k}^R\$ \mid$ where $k \geq 1$ and, for some i, $1 \leq i < 2k$, $C_i \Rightarrow_M C_{i+1}$ is false $\}$,
- $L = L_1 \cup L_2 \cup \{\lambda\}$.

Finite Power Property

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

Another View of Finite Power

- **Create CFGs for the following**
  - L_1 = { #C | C is a configuration of M, a FRS with Residue }
  - L_2 = { $\#C_1\#C_2\#\ldots\#C_n$ | $\sim(C_i \Rightarrow C_{i+1})$, for some I }
  - Consider $L = (\lambda \cup L_1 \cup L_2)$
  - Now, consider L^2.
    - This is $\lambda \cup L_1 \cup L_2 \cup L_1^2 \cup L_2^2 \cup L_1L_2 \cup L_2L_1$
    - But, $L_2^2 \cup L_1L_2 \cup L_2L_1 \subset L_2$
    - So, $L^2 = L \cup L_1^2 = L \cup \{ \#C_1\#C_2 \mid C_1 \Rightarrow C_2 \}$
    - And, $L^k = L \cup L_1^k = L \cup \{ \#C_1\#C_2\#\ldots\#C_k \mid C_i \Rightarrow C_{i+1}, 1 \leq i < k\}$
- **L has the finite power property if and only if M halts in k or fewer steps, for some finite k, independent of its starting configuration. Thus, Finite Power for CFLs is undecidable. Or is this a false proof????**

© UCF (Charles E. Hughes)

Finite Convergence

- **Theorem 6:**
  The problem to determine, for an arbitrary regular language R and context free language L, either of the following predicates is undecidable

  $\exists k \geq 0 \; R\,(k) \triangleright L = R\,(k+1) \triangleright L$

  $\exists k \geq 0 \; R \triangleright^{[k]} L = R \triangleright^{[k+1]} L$

The Magic R and L

- **Let L’ be any arbitrary CFL**
- **It is undecidable if L’ = Σ^***
- **We can check if λ is in L’. If not, L’ $\neq \Sigma^*$**
- **Let L = (L’ #)* L’ and R = Σ^***
- **L’ = Σ^* iff R (0) $\triangleright$ L = R (1) $\triangleright$ L**
  iff $\exists k \geq 0$ R (k) $\triangleright$ L = R (k+1) $\triangleright$ L
- **L’ = Σ^* iff R $\triangleright^{[0]}$ L = R $\triangleright^{[1]}$ L**
  iff $\exists k \geq 0$ R $\triangleright^{[k]}$ L = R $\triangleright^{[k+1]}$ L

© UCF (Charles E. Hughes)

References

- **V. D. Blondel, O. Bournez, P. Koiran, C. H. Papadimitriou, and J. N. Tsitsiklis, Deciding stability and mortality of piecewise affine dynamical systems, *Theoretical Computer Science*** 255(1-2) **(2001) 687-696.**
- **M. Daley, L. Kari, G. Gloor and R. Siromoney, Circular contextual insertions/deletions with applications to biomolecular computation, *String Processing and Information Retrieval Symposium / International Workshop on Groupware*, Cancun, Mexico (1999) 47-54**
- **P. K. Hooper, The undecidability of the Turing machine immortality problem, *Journal of Symbolic Logic*** 31(2) **(1966) 219-234.**
- **C. E. Hughes and S. M. Selkow, The finite power property for context-free languages, *Theoretical Computer Science*** 15(1) **(1981) 111-114.**
- **M. Ito, *Algebraic Theory of Automata and Languages*, World Scientific Publishing Co. Pte. Ltd., Singapore, 2004.**
- **L. Kari, *On insertion and deletion in formal languages*, Ph.D. Thesis, University of Turku, Finland, 1991.**

Propositional Calculus

Axiomatizable Fragments

Propositional Calculus

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

Tautology and Satisfiability

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

Proving Consequences

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

Some Undecidables

- **Given a set of axioms,**
  - Is this set complete?
  - Given a tautology T, is T a consequent?
- **The above are even undecidable with one axiom and with only 2 variables. I will show this result shortly.**

© UCF (Charles E. Hughes)

Refutation

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

© UCF (Charles E. Hughes)

Resolution

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

Axioms

- **Must be tautologies**
- **Can be incomplete**
- **Might have limitations on them and on WFFs, e.g.,**
  - Just implication
  - Only n variables
  - Single axiom

Simulating Machines

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

Diadic PIPC

- **Diadic limits us to two variables**
- **PIPC means Partial Implicational Propositional Calculus, and limits us to implication as only connective**
- **Partial just means we get a fragment**
- **Problems**
  - Is fragment complete?
  - Can F be derived by substitution and MP?

Living without Associativity

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

Composition Encoding

- **Consider $(p \supset p)$, $(p \supset (p \supset p))$, $(p \supset (p \supset (p \supset p)))$, …**
  - No form is a substitution instance of any of the other, so they can't be confused
  - All are tautologies
- **Consider $((X \supset Y) \supset Y)$**
  - This is just $X \vee Y$

© UCF (Charles E. Hughes)

Encoding

- **Use (p ⊃ p) as form of bottom of stack**
- **Use (p ⊃ (p ⊃ p)) as form for letter 0**
- **Use (p ⊃ (p ⊃ (p ⊃ p))) as form for 1**
- **Etc.**
- **String 01 (reading top to bottom of stack) is**
  - (((p ⊃ p) ⊃ ((p ⊃ p) ⊃ ((p ⊃ p) ⊃ (p ⊃ p)))) ⊃
    (((p ⊃ p) ⊃ ((p ⊃ p) ⊃ ((p ⊃ p) ⊃ (p ⊃ p)))) ⊃
    ((p ⊃ p) ⊃ ((p ⊃ p) ⊃ ((p ⊃ p) ⊃ (p ⊃ p))))))

Encodings

$I(p)$ is defined to abbreviate the wff $[p \supset p]$.

$\Phi_0(p)$ is $[p \supset I(p)]$, i.e., $[p \supset [p \supset p]]$.

$\Phi_1(p)$ is $[p \supset \Phi_0(p)]$.

$\xi_1(p)$ is $[p \supset \Phi_1(p)]$.

$\xi_2(p)$ is $[p \supset \xi_1(p)]$.

$\xi_3(p)$ is $[p \supset \xi_2(p)]$.

$\Psi_1(p)$ is $[p \supset \xi_3(p)]$.

$\Psi_2(p)$ is $[p \supset \Psi_1(p)]$.

$\vdots$

$\Psi_n(p)$ is $[p \supset \Psi_{n-1}(p)]$.

Creating Terminal IDs

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

Reversing Print and Left

9. $[\Psi'_k\Phi_i(p_1) \vee p_2] \supset [\Psi'_h\Phi_j(p_1) \vee p_2]$, whenever $q_h a_j a_i q_k \in T$.

10a. $[\Psi'_k\Phi_0 I(p_1) \vee I(p_1)] \supset [\Psi'_h\Phi_0 I(p_1) \vee I(p_1)]$,

b. $[\Psi'_k\Phi_1 I(p_1) \vee I(p_1)] \supset [\Psi'_h\Phi_0 I(p_1) \vee \Phi_1(p_1)]$,

c. $[\Psi'_k\Phi_i I(p_1) \vee \Phi_j(p_2)] \supset [\Psi'_h\Phi_0 I(p_1) \vee \Phi_i\Phi_j(p_2)]$,

d. $[\Psi'_k\Phi_0\Phi_0\Phi_i(p_1) \vee I(p_2)] \supset [\Psi'_h\Phi_0\Phi_i(p_1) \vee I(p_2)]$,

e. $[\Psi'_k\Phi_1\Phi_0\Phi_i(p_1) \vee I(p_2)] \supset [\Psi'_h\Phi_0\Phi_i(p_1) \vee \Phi_1 I(p_2)]$,

f. $[\Psi'_k\Phi_i\Phi_0\Phi_j(p_1) \vee \Phi_m(p_2)] \supset [\Psi'_h\Phi_0\Phi_j(p_1) \vee \Phi_i\Phi_m(p_2)]$,

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

11a. $[\Psi'_k\Phi_0\Phi_1(p_1) \vee I(p_2)] \supset [\Psi'_h\Phi_1(p_1) \vee I(p_2)]$,

b. $[\Psi'_k\Phi_1\Phi_1(p_1) \vee I(p_2)] \supset [\Psi'_h\Phi_1(p_1) \vee \Phi_1 I(p_2)]$,

c. $[\Psi'_k\Phi_i\Phi_1(p_1) \vee \Phi_j(p_2)] \supset [\Psi'_h\Phi_1(p_1) \vee \Phi_i\Phi_j(p_2)]$,

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

© UCF (Charles E. Hughes)

Reversing Right

12a. $[\Psi'_k \Phi_0 I(p_1) \vee I(p_1)] \supset [\Psi'_h \Phi_0 I(p_1) \vee I(p_1)]$,

b. $[\Psi'_k \Phi_0 I(p_1) \vee \Phi_0 \Phi_i(p_2)] \supset [\Psi'_h \Phi_0 I(p_1) \vee \Phi_i(p_2)]$,

c. $[\Psi'_k \Phi_1(p_1) \vee I(p_2)] \supset [\Psi'_h \Phi_0 \Phi_1(p_1) \vee I(p_2)]$,

d. $[\Psi'_k \Phi_0 \Phi_i(p_1) \vee I(p_2)] \supset [\Psi'_h \Phi_0 \Phi_0 \Phi_i(p_1) \vee I(p_2)]$,

e. $[\Psi'_k \Phi_0 \Phi_i(p_1) \vee \Phi_0 \Phi_j(p_2)] \supset [\Psi'_h \Phi_0 \Phi_0 \Phi_i(p_1) \vee \Phi_j(p_2)]$,

f. $[\Psi'_k \Phi_1(p_1) \vee \Phi_0 \Phi_i(p_2)] \supset [\Psi'_h \Phi_0 \Phi_1(p_1) \vee \Phi_i(p_2)]$,

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

13a. $[\Psi'_k \Phi_0 I(p_1) \vee \Phi_1(p_2)] \supset [\Psi'_h \Phi_1 I(p_1) \vee p_2]$,

b. $[\Psi'_k \Phi_1(p_1) \vee \Phi_1(p_2)] \supset [\Psi'_h \Phi_1 \Phi_1(p_1) \vee p_2]$,

c. $[\Psi'_k \Phi_0 \Phi_i(p_1) \vee \Phi_1(p_2)] \supset [\Psi'_h \Phi_1 \Phi_0 \Phi_i(p_1) \vee p_2]$

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

 © UCF (Charles E. Hughes)

The Rest of the Story

- **It's in the paper**
- **Result is that word decision problem for membership in the theorems of a diadic pipc is undecidable**

First Order Predicate Calculus

Undecidability and Reduction Classes

First Order Primitive Symbols

- **Universe of discourse: U**
- **Variables: x, x1, x2, …, y, y1, y2, …, etc. over U**
- **Functions: f, f1, …, g, g1, …, etc. from U^n to U, where n is the arity of the given function**
- **A set of constants denoted a, a1, …, etc. These can be viewed as 0-ary functions.**
- **Predicates: P, P1, …, etc. from U^n to {T,F}.**
- **The logical constants T and F. These can be viewed as 0–ary predicates.**
- **Boolean operators:**
$\wedge$ (and), $\vee$ (or), $\neg$ (not), $\supset$ (implies), $\equiv$ (equivalence)
- **Quantifiers over elements of U: $\exists$ (there exists), $\forall$ (for all)**
- **Braces ([,]) to disambiguate bindings**
- **If we wish, we can also add equality to obtain a first order logic with equality**

First Order Terms

- **Any constant is a term (with no free variables).**
- **Any variable is a term (whose only free variable is itself).**
- **Any expression $f(t1,\ldots,tn)$ of $n \geq 1$ arguments (where each argument ti is a term and f is a function symbol of arity n) is a term. Its free variables are the free variables of any of the terms ti.**
- **Nothing else is a term.**

Well-Formed Formulas (WFFs)

- **If P is a relation of valence $n \geq 1$ and the t_i are terms then $P(t_1,...,t_n)$ is well-formed. Its free variables are the free variables of any of the terms t_i. All such formulas are said to be *atomic*.**
- **If φ is a *wff*, then ¬φ is a *wff*. Its free variables are the free variables of φ.**
- **If φ and ψ are *wffs*, then [φ ∧ ψ], [φ ∨ ψ], [φ ⊃ ψ], [φ ≡ ψ] are *wffs*. Its free variables are the free variables of φ or ψ.**
- **If φ is a *wff*, then ∀x[φ] and ∃x[φ] are *wffs* (and similarly for any other variable in place of *x*). Its free variables are the free variables of φ or ψ other than *x*. Any instance of *x* (or other variable replacing *x* in this construction) is said to be bound — not free — in ∀x[φ] and ∃x[φ].**
- **Nothing else is a *wff*.**

© UCF (Charles E. Hughes)

Substitution

- **If *t* is a term and φ(*x*) is a formula possibly containing *x* as a free variable, then φ(*t*) is defined to be the result of replacing all free instances of *x* by *t*, provided that no free variable of *t* becomes bound in this process.**
- **If some free variable of *t* becomes bound, then to substitute *t* for *x* it is first necessary to change the names of bound variables of φ to something other than the free variables of *t*. To see why this condition is necessary, consider the formula φ(*x*) given by $\forall y\ y \le x$ ("*x* is maximal"). If *t* is a term without *y* as a free variable, then φ(*t*) just means *t* is maximal. However if *t* is *y* the formula φ(*y*) is $\forall y\ y \le y$ which does not say that *y* is maximal. The problem is that the free variable *y* of *t* (=*y*) became bound when we substituted *y* for *x* in φ(*x*). So to form φ(*y*) we must first change the bound variable *y* of φ to something else, say *z*, so that φ(*y*) is then $\forall z\ z \le y$. Forgetting this condition is a notorious cause of errors.**

Inference (Deduction)

- **We can denote deduction by the symbol $\vdash$**
  $\vdash \varphi$ means that φ can be proven with no axioms (pre-suppositions)
  $\pi \vdash \varphi$ means φ can be proven assuming π
  $\pi \vdash \varphi$ is equivalent to $\vdash \pi \supset \varphi$
- **<u>Modus Ponens</u>**
  If φ and $\varphi \supset \psi$ are proved, then one can deduce ψ.
- **<u>Universal Generalization</u>**
  If $\varphi(x)$ is proved then one can deduce $\forall x[\varphi(x)]$
- **<u>Universal Instantiation</u>**
  If $\forall x[\varphi(x)]$ is proved then one can deduce $\varphi(t)$ where all free occurrences of variable x are replaced by the term t. Of course, we can make t = x, and just remove the universal quantifier.

© UCF (Charles E. Hughes)

First Order Theories

- **A first-order theory consists of a finite set of axioms and the statements deducible from them.**
- **In general, it not decidable if a given proposition is deducible within an arbitrary first-order theory. That was proven by Gödel in his famous incompleteness theorem.**

Restricted Forms

- **Prenex:**
  Starts with all quantifiers followed by a quantifier free part, called the matrix
- **Conjunctive Normal Form (CNF):**
  Conjunction (ands) of disjuncts (ors)
  The terms in each disjunct are predicates and negations of predicates

Reduction Classes

- **A Reduction Class is a restricted set of wff, R, such that there exists a total recursive procedure, f, that maps an arbitrary first order wff, w, to a wff, f(w), in R, such that $\vdash$ w iff $\vdash$ f(w).**
- **I will give you a paper that shows that the set of prenex formulas with two universals and no existentials, and in conjunctive normal form with just two variables, x and y, one binary function, f, and one unary predicate, T, is a reduction class.**

Sketch of Proof

- **We first encode variables p1 and p2 as variables x and y.**
- **We then encode $b \supset c$ as $f(b, c)$**
- **We then declare that a formula, P, is a theorem by stating T(P)**
- **A1 is the first axiom. For examples, if axiom A1 is $((p1 \supset p2) \supset p2) \supset ((p2 \supset p1) \supset p1)$ then we encode this as $A1^* = T(f(f(f(x,y),y),f(f(y,x),x)))$**
- **We encode the entire system as D as $A1^* \& \ldots \& An^* \& \forall x \forall y\ [T(x)\ \&\ T(f(x,y)) \supset T(y)]$**

General Idea

- **Get axioms easily from A1*&…&An***
- **Get substitution based on rules from first order such as universal generalization and instantiation**
- **Get MP from the last part**
  $\forall x \forall y\ [T(x)\ \&\ T(f(x,y)) \supset T(y)]$
  which in effect mirrors propositional MP using f as a substitute for implies

Inductive Steps

- **k=1: This can only be an axiom. We can clearly deduce Ai* from D.**
- **k>1:**
  - if axiom, no problem;
  - if substitution of already proved theorem, then do some instantiation and generalization (see paper for details)
  - If MP, also see paper, but it’s really easy; you just must be precise

 © UCF (Charles E. Hughes)

Final Exam Topics

Exclusions as well as Inclusions

Material is from 111 on.

Exclusions

- **No explicit Turing Machines to write**
- **No explicit FRSs to write**
- **No explicit Register Machines to write**
- **No Rice-Shapiro (but Rice is definitely in)**
- **No explicit S-m-n, recursion or fixed point theorems**

© UCF (Charles E. Hughes)

Inclusions (Guarantees)

- **Repeat of material from Exam#2**
- **A question about quantification**
- **A question about Real-Time and/or Finite Power Property**
- **Closure of recursive/re sets**
- **A question about K and/or K_0**
- **Various re and recursive equivalent definitions**
- **A reduction or two; a proof by diagonalization**
- **Use of STP/VALUE**
- **A question about some simple concepts associated with propositional logic**
- **A question about monoids, Post Normal Systems and/or Semi-Thue Systems**
- **More on next page**

Guarantees – More

- Application of Rice's Theorem
- Many-one reduction
- Some CFG that you must write
- Closure question(s)
- Decision problems for languages
- Trace related question
- PCP related question
- Term rewriting question
- L-system question

Sample#1

1. **For each of the following sets, write a set description that involves the use of a minimum sequence of alternating quantifiers in front of a totally computable predicate (typically formed from STP and/or VALUE). Choosing from among (REC) recursive, (RE) re non-recursive, (CO-RE) complement of re non-recursive, (HU) non-re/non-co-re, categorize each of the sets based on the quantified predicate you just wrote. No proofs are required.**

   a.) $S = \{ f \mid f(x) \uparrow \text{ for all } x \}$

   b.) $A = \{ \langle f,x \rangle \mid f(x) = 0 \}$

© UCF (Charles E. Hughes)

Sample#2

2. Let set A be recursive, B be re non-recursive and C be non-re. Choosing from among (REC) recursive, (RE) re non-recursive, (NR) non-re, categorize each of the sets in a) through b) by listing all possible categories. Briefly, but convincingly, justify each answer.

   a.) $A * B = \{ x*y \mid x \in A \text{ and } y \in B \}$

   b.) $B \cap C = \{ x \mid x \in B \text{ and } x \in C \}$

Sample#3

3. Let S be an arbitrary set.

   Show that S is infinite recursive if and only if it can be enumerated by a monotonically increasing function f_S.

Sample#4

4. Prove that the Halting Problem (the set K_0) is not decidable within any formal model of computation.

(Hint: A diagonalization proof is required.)

© UCF (Charles E. Hughes)

Sample#5

5. Consider the set of indices UNDEFINED = { f | $\forall$<x,t> [~STP(x, f, t)]}. Use Rice's Theorem to show that UNDEFINED is not recursive. Hint: There are two properties that must be demonstrated.

Sample#6

6. Show that $\sim K_0 \leq_m$ UNDEFINED, where $\sim K_0 = \{ <f, x> \mid \varphi_f(x)\uparrow = \forall t\, [\sim STP(x, f, t)] \}$.

Sample Question#7

7. **Present a Context-Free Grammar, G, such that L(G) = { $a^i b^j c^k$ | i<k or j<k }.**

What is max(L(G))?

What is min(L(G))?

Sample Question#8

8. **Assuming the undecidability of PCP, show that the ambiguity problem for Context-Free Grammars is undecidable.**

Sample Question#9

9. **Prove that if L is a Context-Free Language then so is $Mid(L) = \{ y \mid \exists x,z [xyz \in L] \}$. You may assume that CFLs are closed under substitution, homomorphism, concatenation, and intersection with regular languages.**

Sample Question#10

10. Let R be regular and L1,L2 be context free. What can you say about the complexity of the languages S?

a) S = L1/R
b) S = L1/L2
c) S = L1∪R
d) S = L1∩L2
e) S, where S⊂R

© UCF (Charles E. Hughes)

Sample Question#11

11. Present an outline of the proof that the CSL's are not closed under homomorphism. You may assume that the phrase structured grammars can produce non-CSL languages.

© UCF (Charles E. Hughes)

Sample Question#12

12. **Why are traces of computation hard (non-CFLs) but the complements of traces are easy (CFLs)?**

Sample Question#13

13. **Categorize the language L as to whether it is a CFL or not. If it is a CFL, show a grammar; if not use the Pumping Lemma to prove this.**
    L =$\{ a^i b^j c^k \mid k \geq i \text{ and } k \geq j \}$

Sample#14

14. Choosing from among (D) decidable, (U) undecidable, (?) unknown, categorize the problem “L is infinite?” for each of the following classes of languages. In each case, justify your answer. You need not provide a proof, but your justification should demonstrate you could do so.

L is Regular:

L is Context Free:

L is Context Sensitive:

L is Recursively Enumerable:

Sample#15

15. Define each of the following:

Satisfiability of a proposition

Immortality Problem for Turing Machines

Sample#16

16. Differentiate Chomsky grammars from Lindenmayer Systems by providing two ways in which they operate differently.

Extra Promises

- **I will create a term rewriting system and an expression that you must rewrite in accordance with the system's rules.**
- **I will create an L-System with an axiom and have you apply the axiom and a second generation. The answer may be in the form of a graphical drawing based on F, + and - operations.**

 © UCF (Charles E. Hughes)

Fibonacci Numbers

If we define the following simple Lindenmayer system (grammar):

non-terminals : A B

constants : none

ω (start symbol) : A

rules : (A → B), (B → AB)

then this L-system produces the following sequence of strings:

n=0 : A

n=1 : B

n=2 :

n=3 :

n=4 :

n=5 :

n=6 :

n=7 :

What is the relation of these strings to Fibonacci numbers?

© UCF (Charles E. Hughes)

Variant of Koch Curve

A variant of the Koch curve which uses only right-angles.

non-terminals : F

constants : + −

start : F

rules : (F → F+F−F−F+F)

Here, *F* means "draw forward", + means "turn left 90°", and − means "turn right 90°". Write the strings and draw the images associated with the following numbers of iterations. I did n=0.

n=0:

F **−**

n=1:

n=2:

Recursion Theorem

Self Reproducibility

Simple Form

Theorem: There is an index e, such that

$\forall x\ \phi_e(x) = e$

- **This means that we have a function that always produces its own description (index) no matter what input you give it.**
- **People used to have fun trying to find the smallest self-reproducing Lisp program or Turing machine.**

Fixed Point Theorem

A property of all indexing schemes

The Fixed Point Theorem

Theorem: Let f(z) be any computable function. Then there is an index e such that

$$\forall x \; \phi_{f(e)}(x) = \phi_e(x)$$

- **There are many forms of computation that seek a fixed point. Correctness proofs are often of this sort.**

© UCF (Charles E. Hughes)

Classifying Unsolvable Problems

Rice-Shapiro Theorem
Minimum Quantification

Rice-Shapiro Theorem

- **Properties of a set of indices P that are required if P is re:**
  - If L is in (has property) P and $L \subseteq L'$, for some re set L', then L' is in P.
  - If L is an infinite set in P, then there is some finite subset L' of L that is in P.
  - The set of finite languages in P is enumerable.

Using Extended Rice

- **Violate Condition # 1**
  - $L = \emptyset$
  - L is recursive
  - L is a singleton set
  - L is a regular set
- **Violates Condition # 2**
  - $L = \Sigma^*$
- **Violates Condition # 3**
  - $L - L_u \neq \emptyset$

 © UCF (Charles E. Hughes)

RE Sets

- **$L \neq \varnothing$**

- **L contains at least 3 numbers**

- **W is in L, for a fixed W**

- **$L \cap L_u \neq \varnothing$**

© UCF (Charles E. Hughes)

Minimum Quantification

- **Recursive: unquantified total predicate**
- **RE: existentially quantified**
- **~ RE: universally quantified**
- **Recursive: Can express as RE & ~RE**
- **TOT: universal/existential**
- **~TOT: existential/universal**