CSE 311: Foundations of Computing

Fall 2014

Lecture 30: Wrap up

announcements

- Hand in Homework 9 now
  - Pick up all old homework and exams now
  - Solutions will be available on-line (username 311 password Turing) by tomorrow.
- Review sessions
  - Saturday and Sunday, 4pm, EEB 125
  - List of Final Exam Topics, Practice Final and sample exam questions linked on the web
  - Bring your questions to the review session!
- Final exam
  - Monday, 2:30-4:20 pm or 4:30-6:20, Kane 210
  - Fill in Catalyst Survey by Sunday, 3pm to choose.

General phenomenon: can't tell a book by its cover

and you can't tell what a program does just by its code…

Rice's Theorem: In general there is no way to tell anything about the input/output (I/O) behavior of a program P just given it code!

Note: The statement above is not precise, and we didn't prove it, so this isn't something you can use on homework or exams

Even harder problems

- With the halting problem, by using the Universal machine (a program interpreter) we can simulate P and input x and always get the true answers correct
  - we can't be sure about answering false
- For other problems we can always answer false correctly but maybe not the true answers
- There are natural problems where you can't even do that!
  - The EQUIV problem is an example of this kind of even harder problem

Quick lessons

- Don't rely on the idea of improved compilers and programming languages to eliminate major programming errors
  - truly safe languages can't possibly do general computation
- Document your code!!!!
  - there is no way you can expect someone else to figure out what your program does with just your code ....since....in general it is provably impossible to do this!

CSE 311: Foundations of Computing

Fall 2014

The "5 minute" version

about the course

- From the CSE catalog:
  - **CSE 311 Foundations of Computing I (4)** Examines fundamentals of logic, set theory, induction, and algebraic structures with applications to computing; finite state machines; and limits of computability. Prerequisite: CSE 143; either MATH 126 or MATH 136.
- What this course is about:
  - Foundational structures for the practice of computer science and engineering

propositional logic

- Statements with truth values
  - The Washington State flag is red
  - It snowed in Whistler, BC on January 4, 2011.
  - Rick Perry won the Iowa straw poll
  - Space aliens landed in Roswell, New Mexico
  - If n is an integer greater than two, then the equation $a^n + b^n = c^n$ has no solutions in non-zero integers a, b, and c.
- Propositional variables: *p, q, r, s, . . .*
- Truth values: **T** for true, **F** for false
- Compound propositions

Negation (not)	$\neg p$
Conjunction (and)	$p \wedge q$
Disjunction (or)	$p \vee q$
Exclusive or	$p \oplus q$
Implication	$p \rightarrow q$
Biconditional	$p \leftrightarrow q$

logical equivalence

- Terminology: A compound proposition is a
  - *Tautology* if it is always true
  - *Contradiction* if it is always false
  - *Contingency* if it can be either true or false

$p \vee \neg p$

$p \oplus p$

$(p \rightarrow q) \wedge p$

$(p \wedge q) \vee (p \wedge \neg q) \vee (\neg p \wedge q) \vee (\neg p \wedge \neg q)$

logical equivalence

- p and q are *logically equivalent* iff $p \leftrightarrow q$ is a tautology
- The notation $p \equiv q$ denotes p and q are logically equivalent

- De Morgan's Laws:

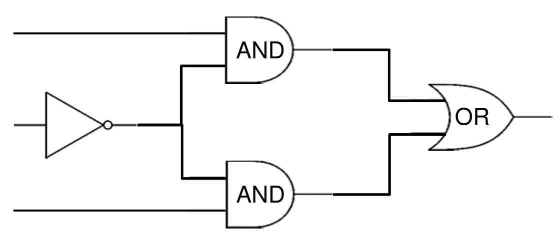
$$\neg (p \wedge q) \equiv \neg p \vee \neg q$$
$$\neg (p \vee q) \equiv \neg p \wedge \neg q$$

digital circuits

- Computing with logic
  - T corresponds to 1 or "high" voltage
  - F corresponds to 0 or "low" voltage

- Gates
  - Take inputs and produce outputs
    Functions
  - Several kinds of gates
  - Correspond to propositional connectives
    Only symmetric ones (order of inputs irrelevant)

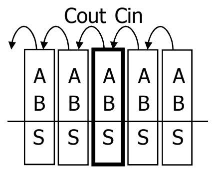
combinational logic circuits



Wires can send one value to multiple gates

a simple example: 1-bit binary adder

- Inputs: A, B, Carry-in
- Outputs: Sum, Carry-out



A	B	Cin	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Cout = B Cin + A Cin + A B

S = A′ B′ Cin + A′ B Cin′ + A B′ Cin′ + A B Cin

Truth Tables to Boolean Logic

c3 = (DAY == SUN and LEC) or (DAY == MON and LEC)

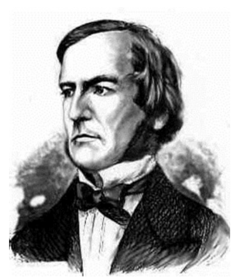
c3 = (d2 == 0 && d1 == 0 && d0 == 0 && L == 1) ||
(d2 == 0 && d1 == 0 && d0 == 1 && L == 1)

c3 = d2'•d1'•d0'•L + d2'•d1'•d0•L

DAY	d2d1d0	L	c0	c1	c2	c3
SunS	000	0	0	1	0	0
SunL	000	1	0	0	0	1
MonS	001	0	0	1	0	0
MonL	001	1	0	0	0	1
TueS	010	0	0	1	0	0
TueL	010	1	0	0	1	0
WedS	011	0	0	1	0	0
WedL	011	1	0	0	1	0
Thu	100	-	0	1	0	0
FriS	101	0	1	0	0	0
FriL	101	1	0	1	0	0
Sat	110	-	1	0	0	0
-	111	-	-	-	-	-

Boolean Algebra

- Boolean algebra to circuit design
- Boolean algebra
  - a set of elements B containing {0, 1}
  - binary operations { + , • }
  - and a unary operation { ' }
  - such that the following axioms hold:



1. the set B contains at least two elements: 0, 1

For any a, b, c in B:

2. closure:	a + b is in B	a • b is in B
3. commutativity:	a + b = b + a	a • b = b • a
4. associativity:	a + (b + c) = (a + b) + c	a • (b • c) = (a • b) • c
5. identity:	a + 0 = a	a • 1 = a
6. distributivity:	a + (b • c) = (a + b) • (a + c)	a • (b + c) = (a • b) + (a • c)
7. complementarity:	a + a' = 1	a • a' = 0

sum-of-products canonical forms

- Also known as disjunctive normal form
- Also known as minterm expansion

F = *001* *011* *101* *110* *111*

F = A'B'C + A'BC + AB'C + ABC' + ABC

A	B	C	F	F'
0	0	0	0	1
0	0	1	1	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	0

F' = A'B'C' + A'BC' + AB'C'

Predicate Logic

Predicate or **Propositional Function**

– A function that returns a truth value, e.g.,

"x is a cat"

"x is prime"

"student x has taken course y"

"$x > y$"

"$x + y = z$" or Sum(x, y, z)

"$5 < x$"

Predicates will have **variables** or **constants** as arguments.

statements with quantifiers

Domain:
Positive Integers

Even(x)
Odd(x)
Prime(x)
Greater(x,y)
Equal(x,y)

- $\forall x\ (\text{Even}(x) \vee \text{Odd}(x))$
- $\exists x\ (\text{Even}(x) \wedge \text{Prime}(x))$
- $\forall x\ \exists y\ (\text{Greater}(y, x) \wedge \text{Prime}(y))$
- $\forall x\ (\text{Prime}(x) \rightarrow (\text{Equal}(x, 2) \vee \text{Odd}(x))$
- $\exists x\ \exists y (\text{Equal}(x, y + 2) \wedge \text{Prime}(x) \wedge \text{Prime}(y))$

English to Predicate Logic

Cat(x)
Red(x)
LikesTofu(x)

- "Red cats like tofu"

- "Some red cats don't like tofu"

De Morgan's laws for Quantifiers

$$\neg\forall x\ P(x) \equiv \exists x\ \neg P(x)$$
$$\neg\exists x\ P(x) \equiv \forall x\ \neg P(x)$$

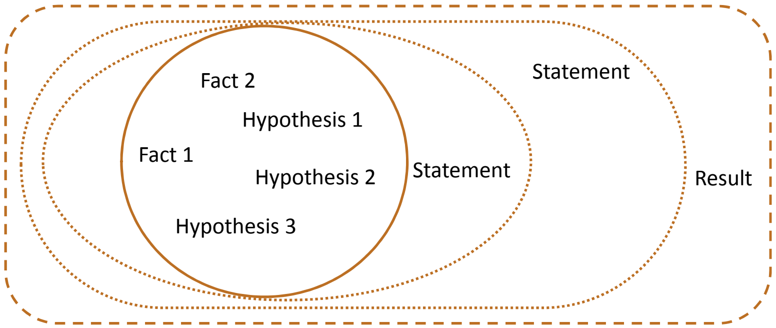
"There is no largest integer"

$$\neg\exists x\ \forall y\ (x \geq y)$$
$$\equiv \forall x\ \neg\forall y\ (x \geq y)$$
$$\equiv \forall x\ \exists y\ \neg(x \geq y)$$
$$\equiv \forall x\ \exists y\ (y > x)$$

"For every integer there is a larger integer"

Proofs

- Start with hypotheses and facts
- Use rules of inference to extend set of facts
- Result is proved when it is included in the set



Simple Propositional Inference Rules

Excluded middle plus two inference rules per binary connective, one to eliminate it and one to introduce it

$$\frac{p \land q}{\therefore\ p,\ q} \qquad \frac{p,\ q}{\therefore\ p \land q}$$

$$\frac{p \lor q\ ,\ \neg p}{\therefore\ q} \qquad \frac{p}{\therefore\ p \lor q,\ q \lor p}$$

$$\frac{p,\ p \to q}{\therefore\ q} \qquad \frac{p \Rightarrow q}{\therefore\ p \to q}$$

Direct Proof Rule

Not like other rules

Inference rules for quantifiers

$$\frac{P(c) \text{ for some } c}{\therefore\ \exists x\ P(x)} \qquad \frac{\forall x\ P(x)}{\therefore\ P(a) \text{ for any } a}$$

$$\frac{\text{“Let } a \text{ be anything*”} \ldots P(a)}{\therefore\ \forall x\ P(x)} \qquad \frac{\exists x\ P(x)}{\therefore\ P(c) \text{ for some } special\text{** } c}$$

* in the domain of P

** By special, we mean that c is a name for a value where P(c) is true. We can't use anything else about that value, so c has to be a NEW variable!

even and odd

$Even(x) \equiv \exists y\ \ (x=2y)$
$Odd(x) \equiv \exists y\ \ (x=2y+1)$
Domain: Integers

- **Prove: “The square of every odd number is odd”**
  English proof of: $\forall x\ (Odd(x) \to Odd(x^2))$

Let x be an odd number.
Then $x=2k+1$ for some integer k (depending on x)
Therefore $x^2=(2k+1)^2= 4k^2+4k+1=2(2k^2+2k)+1$.
Since $2k^2+2k$ is an integer, x^2 is odd.

Definitions

- A and B are *equal* if they have the same elements

$$A = B \equiv \forall x\, (x \in A \leftrightarrow x \in B)$$

- A is a *subset* of B if every element of A is also in B

$$A \subseteq B \equiv \forall x\, (x \in A \rightarrow x \in B)$$

- Note: $\boldsymbol{(A = B) \equiv (A \subseteq B) \wedge (B \subseteq A)}$

Set Operations

$A \cup B = \{ x : (x \in A) \vee (x \in B) \}$ **Union**

$A \cap B = \{ x : (x \in A) \wedge (x \in B) \}$ **Intersection**

$A \setminus B = \{ x : (x \in A) \wedge (x \notin B) \}$ **Set Difference**

$A \oplus B = \{ x : (x \in A) \oplus (x \in B) \}$ **Symmetric Difference**

$\overline{A} = \{ x : x \notin A \}$ (with respect to universe U) **Complement**

Empty Set, Power set, Cartesian Product

- *Power set* of a set A = set of all subsets of A

$$\mathcal{P}(A) = \{ B : B \subseteq A \}$$

e.g. $\text{Days} = \{M, W, F\}$

$$\mathcal{P}(\text{Days}) = \{ \varnothing, \{M\}, \{W\}, \{F\}, \{M,W\}, \{W,F\}, \{M,F\}, \{M,W,F\} \}$$

e.g. $\mathcal{P}(\varnothing) = ?$

$$A \times B = \{ (a,b) : a \in A, b \in B \}$$

Bitwise Operations

$$\begin{array}{rl} & 01101101 \\ \vee & \underline{00110111} \\ & 01111111 \end{array}$$

Java: `z=x|y`

$$\begin{array}{rl} & 00101010 \\ \wedge & \underline{00001111} \\ & 00001010 \end{array}$$

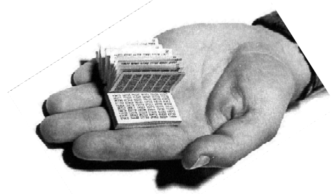
Java: `z=x&y`

$$\begin{array}{rl} & 01101101 \\ \oplus & \underline{00110111} \\ & 01011010 \end{array}$$

Java: `z=x^y`

One-Time Pad

- Alice and Bob privately share random n-bit vector K
  - Eve does not know K
- Later, Alice has n-bit message m to send to Bob
  - Alice computes $C = m \oplus K$
  - Alice sends C to Bob
  - Bob computes $m = C \oplus K$ which is $(m \oplus K) \oplus K$
- Eve cannot figure out m from C unless she can guess K



division theorem

Let a be an integer and d a positive integer. Then there are *unique* integers q and r, with $0 \leq r < d$, such that $a = dq + r$.

$q = a \textbf{ div } d \qquad r = a \textbf{ mod } d$

Arithmetic, mod 7

$$a +_7 b = (a + b) \bmod 7$$
$$a \times_7 b = (a \times b) \bmod 7$$

+	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	1	2	3	4	5	6	0
2	2	3	4	5	6	0	1
3	3	4	5	6	0	1	2
4	4	5	6	0	1	2	3
5	5	6	0	1	2	3	4
6	6	0	1	2	3	4	5

x	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6
2	0	2	4	6	1	3	5
3	0	3	6	2	5	1	4
4	0	4	1	5	2	6	3
5	0	5	3	1	6	4	2
6	0	6	5	4	3	2	1

modular arithmetic

Let a and b be integers, and m be a positive integer. We say a *is congruent to b modulo m* if m divides a – b. We use the notation $a \equiv b \pmod{m}$ to indicate that a is congruent to b modulo m.

Let a and b be integers, and let m be a positive integer. Then $a \equiv b \pmod{m}$ if and only if a mod m = b mod m.

Let m be a positive integer. If $a \equiv b \pmod{m}$ and $c \equiv d \pmod{m}$, then

$a + c \equiv b + d \pmod{m}$ and

$ac \equiv bd \pmod{m}$

Let a and b be integers, and let m be a positive integer. Then $a \equiv b \pmod{m}$ if and only if a mod m = b mod m.

Two's Complement Representation

n bit signed integers, first bit will still be the sign bit

Suppose $0 \le x < 2^{n-1}$,
x is represented by the binary representation of x
Suppose $0 \le x \le 2^{n-1}$,
$-x$ is represented by the binary representation of $2^n - x$

Key property: Twos complement representation of any number y is equivalent to y mod 2^n so arithmetic works mod 2^n

99 = 64 + 32 + 2 + 1
18 = 16 + 2

For n = 8:
99: 0110 0011
-18: 1110 1110

hashing

- Map values from a large domain, 0…M-1 in a much smaller domain, 0…n-1
- Index lookup
- Test for equality
- Hash(x) = x mod p
  - (or Hash(x) = (ax + b) mod p)
- Often want the hash function to depend on all of the bits of the data
  - Collision management

modular exponentiation mod 7

x	1	2	3	4	5	6
1	1	2	3	4	5	6
2	2	4	6	1	3	5
3	3	6	2	5	1	4
4	4	1	5	2	6	3
5	5	3	1	6	4	2
6	6	5	4	3	2	1

a	a^1	a^2	a^3	a^4	a^5	a^6
1	1	1	1	1	1	1
2	2	4	1	2	4	1
3	3	2	6	4	5	1
4	4	2	1	4	2	1
5	5	4	6	2	3	1
6	6	1	6	1	6	1

Repeated Squaring – small and fast

Since a mod m ≡ a (mod m) for any a

we have a^2 mod m = (a mod m)^2 mod m
and a^4 mod m = (a^2 mod m)^2 mod m
and a^8 mod m = (a^4 mod m)^2 mod m
and a^{16} mod m = (a^8 mod m)^2 mod m
and a^{32} mod m = (a^{16} mod m)^2 mod m

Can compute a^k mod m **for** $k=2^i$ **in only i steps**

Primality

An integer p greater than 1 is called *prime* if the only positive factors of p are 1 and p.

A positive integer that is greater than 1 and is not prime is called *composite*.

Fundamental Theorem of Arithmetic

Every positive integer greater than 1 has a unique prime factorization

48 = 2 • 2 • 2 • 2 • 3
45,523 = 45,523
321,950 = 2 • 5 • 5 • 47 • 137
1,234,567,890 = 2 • 3 • 3 • 5 • 3,607 • 3,803

Euclid's Theorem

There are an infinite number of primes.

Proof by contradiction:

Suppose that there are only a finite number of primes: $p_1, p_2, \ldots, p_n$

GCD and Factoring

$a = 2^3 \cdot 3 \cdot 5^2 \cdot 7 \cdot 11 = 46{,}200$

$b = 2 \cdot 3^2 \cdot 5^3 \cdot 7 \cdot 13 = 204{,}750$

$\mathrm{GCD}(a, b) = 2^{\min(3,1)} \cdot 3^{\min(1,2)} \cdot 5^{\min(2,3)} \cdot 7^{\min(1,1)} \cdot 11^{\min(1,0)} \cdot 13^{\min(0,1)}$

Factoring is expensive!

Can we compute GCD(a,b) without factoring?

Euclid's Algorithm

GCD(x, y) = GCD(y, x mod y)

```
int GCD(int a, int b){ /* a >= b, b > 0 */
    int tmp;
    while (b > 0) {
        tmp = a % b;
        a = b;
        b = tmp;
    }
    return a;
}
```

Example: GCD(660, 126)

Extended Euclidean algorithm

- **Can use Euclid's Algorithm to find s, t such that** $\gcd(a,b) = sa + tb$
- e.g. gcd(35,27):

35 = *1* • 27 + 8	35 - *1* • 27 = 8
27= *3* • 8 + 3	27- *3* • 8 = 3
8 = *2* • 3 + 2	8 - *2* • 3 = 2
3 = *1* • 2 + 1	3 - *1* • 2 = 1
2 = *2* • 1 + 0	

- **Substitute back from the bottom**

1= 3 - *1* • 2 = 3 – *1* (8 - *2* • 3) = (-*1*) • 8 + *3* • 3
= (-*1*) • 8 + *3* (27- *3* • 8) = *3* • 27 + (-*10*) • 8
= *3* • 27 + (-*10*) • (35 - *1* • 27) = (-10) • 35 + 13 • 27

Solving Modular Equations

Solving $ax \equiv b \pmod{m}$ for unknown x when $\gcd(a,m) = 1$.

1. **Find s such that $sa + tm = 1$**
2. **Compute $a^{-1} = s \bmod m$, the multiplicative inverse of a modulo m**
3. **Set $x = (a^{-1} \cdot b) \bmod m$**

Mathematical Induction

$$\frac{\begin{array}{l} P(0) \\ \forall\, k\, (P(k) \;\rightarrow\; P(k+1)) \end{array}}{\therefore\, \forall\, n\, P(n)}$$

1. Prove P(0)		**Base Case**
2. Let k be an arbitrary integer ≥ 0		**Inductive Hypothesis**
3. Assume that P(k) is true		
4. ...		**Inductive Step**
5. Prove P(k+1) is true		
6. P(k) → P(k+1)	Direct Proof Rule	**Conclusion**
7. ∀ k (P(k) → P(k+1))	Intro ∀ from 2-6	
8. ∀ n P(n)	Induction Rule 1&7	

strong induction

$$\frac{\begin{array}{l} P(0) \\ \forall k\, \big((P(0) \wedge P(1) \wedge P(2) \wedge \cdots \wedge P(k)) \rightarrow P(k+1) \big) \end{array}}{\therefore\, \forall n\, P(n)}$$

1. **By induction we will show that $P(n)$ is true for every $n \geq 0$**
2. **Base Case: Prove $P(0)$**
3. **Inductive Hypothesis:**
   Assume that for some arbitrary integer $k \geq 0$, $P(j)$ is true for every j from 0 to k
4. **Inductive Step:**
   Prove that $P(k+1)$ is true using the Inductive Hypothesis (that $P(j)$ is true for all values $\leq k$)
5. **Conclusion: Result follows by induction**

5 Steps To Inductive Proofs In English

Proof:

1. "We will show that P(n) is true for every n ≥ 0 by Induction."
2. "Base Case:" Prove P(0)
3. "Inductive Hypothesis:"
   Assume P(k) is true for some arbitrary integer k ≥ 0"
4. "Inductive Step:" Want to prove that P(k+1) is true:
   Use the goal to figure out what you need.
   Make sure you are using I.H. and point out where you are using it. (Don't assume P(k+1) !!)
5. "Conclusion: Result follows by induction"

Strong Induction

$$\frac{\begin{array}{l} P(0) \\ \forall k\ \big((P(0) \land P(1) \land P(2) \land \cdots \land P(k)) \rightarrow P(k+1)\big) \end{array}}{\therefore \forall n\ P(n)}$$

Follows from ordinary induction applied to
$Q(n) = P(0) \land P(1) \land P(2) \land \cdots \land P(n)$

strong induction english proofs

1. By induction we will show that $P(n)$ is true for every $n \geq 0$
2. Base Case: Prove $P(0)$
3. Inductive Hypothesis:
   Assume that for some arbitrary integer $k \geq 0$, $P(j)$ is true for every j from 0 to k
4. Inductive Step:
   Prove that $P(k+1)$ is true using the Inductive Hypothesis (that $P(j)$ is true for all values $\leq k$)
5. Conclusion: Result follows by induction

recursive definitions of functions

- $F(0) = 0$; $F(n+1) = F(n) + 1$ for all $n \geq 0$
- $G(0) = 1$; $G(n+1) = 2 \times G(n)$ for all $n \geq 0$
- $0! = 1$; $(n+1)! = (n+1) \times n!$ for all $n \geq 0$
- $H(0) = 1$; $H(n+1) = 2^{H(n)}$ for all $n \geq 0$

Strings

- An *alphabet* Σ is any finite set of characters
- The set Σ^* of *strings* over the alphabet Σ is defined by
  - **Basis:** $\varepsilon \in \Sigma^*$ (ε is the empty string)
  - **Recursive:** if $w \in \Sigma^*$, $a \in \Sigma$, then $wa \in \Sigma^*$

Function Definitions on Recursively Defined Sets

Length:

len (ε) = 0;

len (wa) = 1 + len(w); for $w \in \Sigma^*, a \in \Sigma$

Reversal:

$\varepsilon^{\mathrm{R}} = \varepsilon$

$(wa)^{\mathrm{R}} = aw^{\mathrm{R}}$ for $w \in \Sigma^*$, $a \in \Sigma$

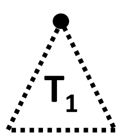
Concatenation:

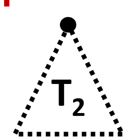
$x \bullet \varepsilon = x$ for $x \in \Sigma^*$

$x \bullet wa = (x \bullet w)a$ for $x, w \in \Sigma^*$, $a \in \Sigma$

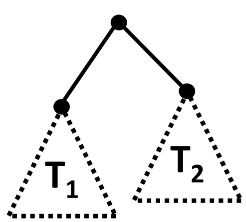
Rooted Binary Trees

- **Basis:** • is a rooted binary tree
- **Recursive step:**

If 

and 

are rooted binary trees,

then so is: 

Structural Induction

How to prove $\forall\, x \in S, P(x)$ is true:

Base Case: Show that $P(u)$ is true for all specific elements u of S mentioned in the *Basis step*

Inductive Hypothesis: Assume that P is true for some arbitrary values of *each* of the existing named elements mentioned in the *Recursive step*

Inductive Step: Prove that $P(w)$ holds for each of the new elements w constructed in the *Recursive step* using the named elements mentioned in the Inductive Hypothesis

Conclude that $\forall\, x \in S, P(x)$

Function Definitions on Recursively Defined Sets

Length:

$len(\varepsilon) = 0$

$len(wa) = 1 + len(w)$ for $w \in \Sigma^*$, $a \in \Sigma$

Reversal:

$\varepsilon^R = \varepsilon$

$(wa)^R = aw^R$ for $w \in \Sigma^*$, $a \in \Sigma$

Concatenation:

$x \bullet \varepsilon = x$ for $x \in \Sigma^*$

$x \bullet wa = (x \bullet w)a$ for $x \in \Sigma^*$, $a \in \Sigma$

Number of c's in a string:

$\#_c(\varepsilon) = 0$

$\#_c(wc) = \#_c(w) + 1$ for $w \in \Sigma^*$

$\#_c(wa) = \#_c(w)$ for $w \in \Sigma^*$, $a \in \Sigma$, $a \neq c$

Regular Expressions

Regular expressions over Σ

- **Basis:**
  - $\varnothing$, ε are regular expressions
  - ***a*** is a regular expression for any $a \in \Sigma$
- **Recursive step:**
  - If **A** and **B** are regular expressions then so are:
    - **(A ∪ B)**
    - **(AB)**
    - **A***

regular expressions

- ***0****
- ***0*1****
- ***(0 ∪ 1)****
- ***(0*1*)****
- ***(0 ∪ 1)* 0110 (0 ∪ 1)****
- ***(0 ∪ 1)* (0110 ∪ 100)(0 ∪ 1)****

Examples

- ***0****
- ***0*1****
- ***(0 ∪ 1)0(0 ∪ 1)0***
- ***(0*1*)****
- ***(0 ∪ 1)* 0110 (0 ∪ 1)****
- ***(00 ∪ 11)* (01010 ∪ 10001)(0 ∪ 1)****

Context-Free Grammars

Example: $S \rightarrow 0S0 \mid 1S1 \mid 0 \mid 1 \mid \varepsilon$

Example: $S \rightarrow 0S \mid S1 \mid \varepsilon$

Context-Free Grammars

Grammar for $\{0^n 1^n : n \geq 0\}$

(all strings with same # of 0's and 1's with all 0's before 1's)

Example: $S \rightarrow (S) \mid SS \mid \varepsilon$

building precedence in simple arithmetic expressions

- **E** – expression (start symbol)
- **T** – term **F** – factor **I** – identifier **N** - number

$E \rightarrow T \mid E+T$

$T \rightarrow F \mid F*T$

$F \rightarrow (E) \mid I \mid N$

$I \rightarrow x \mid y \mid z$

$N \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

definitions for relations

Let A and B be sets,
A binary relation from A to B is a subset of $A \times B$

Let A be a set,
A binary relation on A is a subset of $A \times A$

Let R be a relation on A

R is reflexive iff $(a,a) \in R$ for every $a \in A$

R is symmetric iff $(a,b) \in R$ implies $(b,a) \in R$

R is antisymmetric iff $(a,b) \in R$ and $a \neq b$ implies $(b,a) \notin R$

R is transitive iff $(a,b) \in R$ and $(b,c) \in R$ implies $(a,c) \in R$

Combining Relations

Let R be a relation from A to B.
Let S be a relation from B to C.

The composition of R and S, $S \circ R$ is the relation from A to C defined by:

$S \circ R = \{(a, c) \mid \exists$ b such that $(a,b) \in R$ and $(b,c) \in S\}$

Intuitively, a pair is in the composition if there is a "connection" from the first to the second.

Relations

$(a,b) \in$ Parent: b is a parent of a
$(a,b) \in$ Sister: b is a sister of a
Aunt = Sister ° Parent
Grandparent = Parent ° Parent

$R^2 = R \circ R = \{(a, c) \mid \exists$ b such that $(a,b) \in R$ and $(b,c) \in R\}$

$R^0 = \{(a,a) \mid a \in A\}$
$R^1 = R$
$R^{n+1} = R^n \circ R$

$S \circ R = \{(a, c) \mid \exists$ b such that $(a,b) \in R$ and $(b,c) \in S\}$

n-ary relations

Let $A_1, A_2, \ldots, A_n$ be sets. An n-ary relation on these sets is a subset of $A_1 \times A_2 \times \ldots \times A_n$.

Student_ID	Name	GPA
328012098	Knuth	4.00
481080220	Von Neuman	3.78
238082388	Russell	3.85
238001920	Einstein	2.11
1727017	Newton	3.61
348882811	Karp	3.98
2921938	Bernoulli	3.21
2921939	Bernoulli	3.54

Student_ID	Major
328012098	CS
481080220	CS
481080220	Mathematics
238082388	Philosophy
238001920	Physics
1727017	Mathematics
348882811	CS
1727017	Physics
2921938	Mathematics
2921939	Mathematics

matrix representation for relations

Relation R on $A=\{a_1, \ldots a_p\}$

$$m_{ij} = \begin{cases} 1 \text{ if } (a_i, a_j) \in R, \\ 0 \text{ if } (a_i, a_j) \notin R. \end{cases}$$

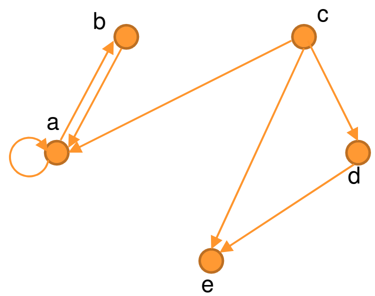
{(1, 1), (1, 2), (1, 4), (2,1), (2,3), (3,2), (3, 3) (4,2) (4,3)}

$$\begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

representation of relations

Directed Graph Representation (Digraph)

{(a, b), (a, a), (b, a), (c, a), (c, d), (c, e) (d, e) }



Connectivity In Graphs

Let R be a relation on a set A. There is a path of length **k** from a to b if and only if (a,b) $\in R^k$

Two vertices in a graph are connected iff there is a path between them.

Let R be a relation on a set A. The connectivity relation R* consists of the pairs (a,b) such that there is a path from a to b in R.

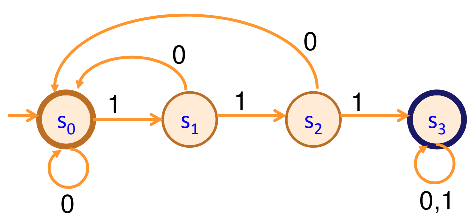
$$R^* = \bigcup_{k=0}^{\infty} R^k$$

Note: The Rosen text uses the wrong definition of this quantity. What the text defines (ignoring k=0) is usually called R^+

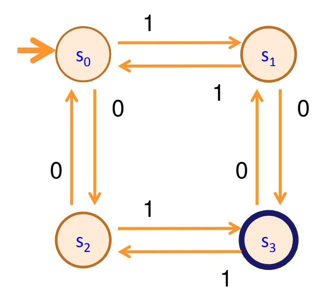
Finite State Machines

- States
- Transitions on inputs
- Start state and final states
- The language recognized by a machine is the set of strings that reach a final state

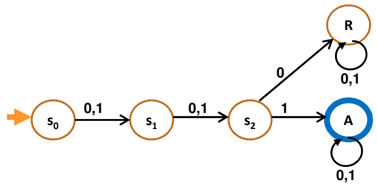
State	0	1
s_0	s_0	s_1
s_1	s_0	s_2
s_2	s_0	s_3
s_3	s_3	s_3

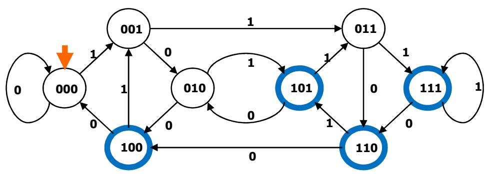


accept strings with odd # of 1's and odd # of 0's



The beginning versus the end

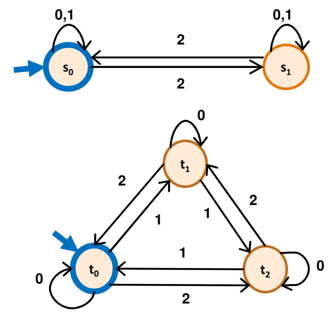


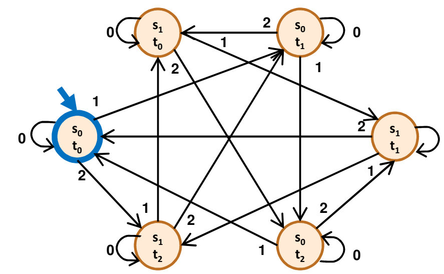


product construction

– **Combining FSMs to check two properties at once**

New states record states of both FSMs

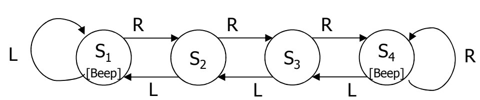




State Machines with Output

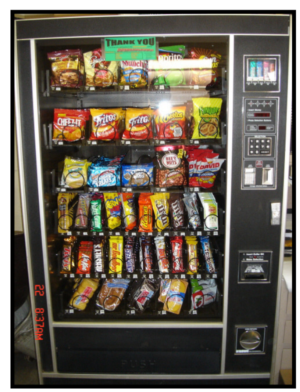
	Input		Output
State	L	R	
s_1	s_1	s_2	Beep
s_2	s_1	s_3	
s_3	s_2	s_4	
s_4	s_3	s_4	Beep

"Tug-of-war"

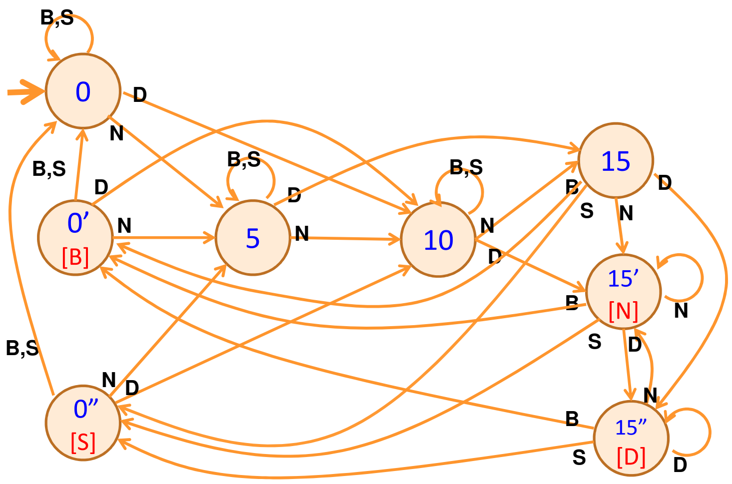


Vending Machine

Enter 15 cents in dimes or nickels
Press S or B for a candy bar



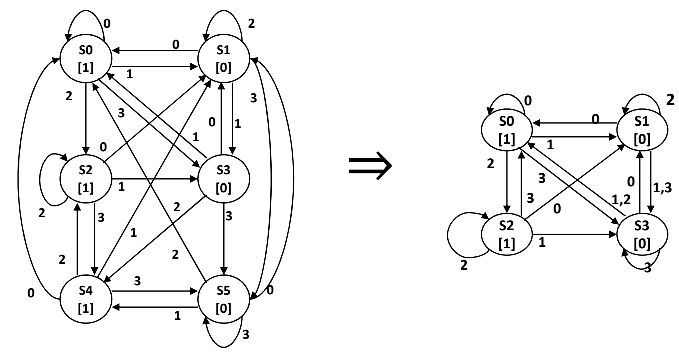
Vending Machine, v1.0



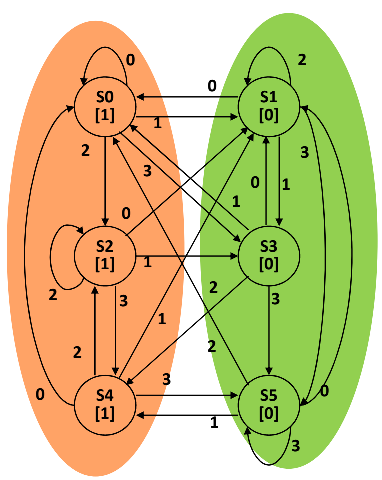
Adding additional “unexpected” transitions

state minimization

Finite State Machines with output at states



state minimization example

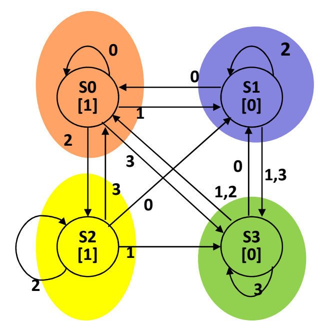


present state	next state 0	1	2	3	output
S0	S0	S1	S2	S3	1
S1	S0	S3	S1	S5	0
S2	S1	S3	S2	S4	1
S3	S1	S0	S4	S5	0
S4	S0	S1	S2	S5	1
S5	S1	S4	S0	S5	0

state
transition table

Put states into groups based on their outputs (or whether they are final states or not)

minimized machine



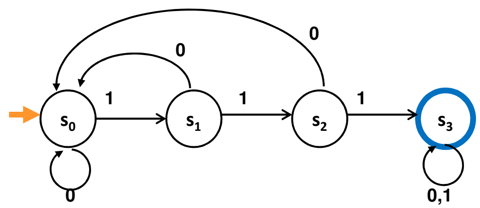
present state	next state 0	1	2	3	output
S0	S0	S1	S2	S3	1
S1	S0	S3	S1	S3	0
S2	S1	S3	S2	S0	1
S3	S1	S0	S0	S3	0

state
transition table

another way to look at DFAs

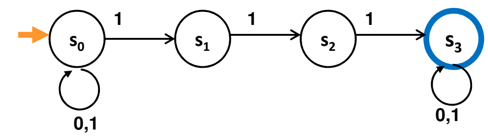
Definition: The label of a path in a DFA is the concatenation of all the labels on its edges in order

Lemma: x is in the language recognized by a DFA iff x labels a path from the start state to some final state

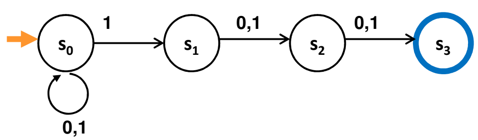


nondeterministic finite automaton (NFA)

- Graph with start state, final states, edges labeled by symbols (like DFA) but
  - Not required to have exactly 1 edge out of each state labeled by each symbol— can have 0 or >1
  - Also can have edges labeled by empty string ε
- **Definition:** x is in the language recognized by an NFA if and only if x labels a path from the start state to some final state



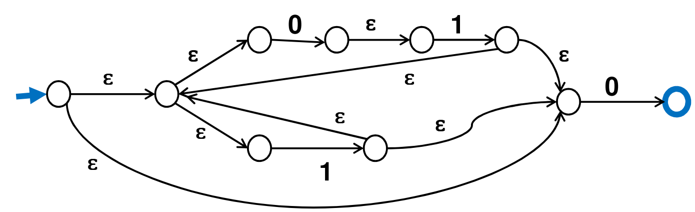
nondeterministic finite automaton



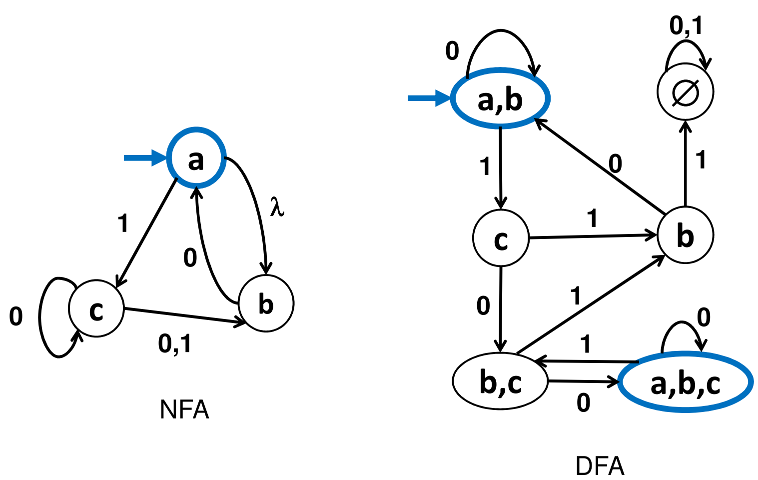
Accepts strings with a 1 three positions from the end of the string

Building an NFA from a Regular Expression

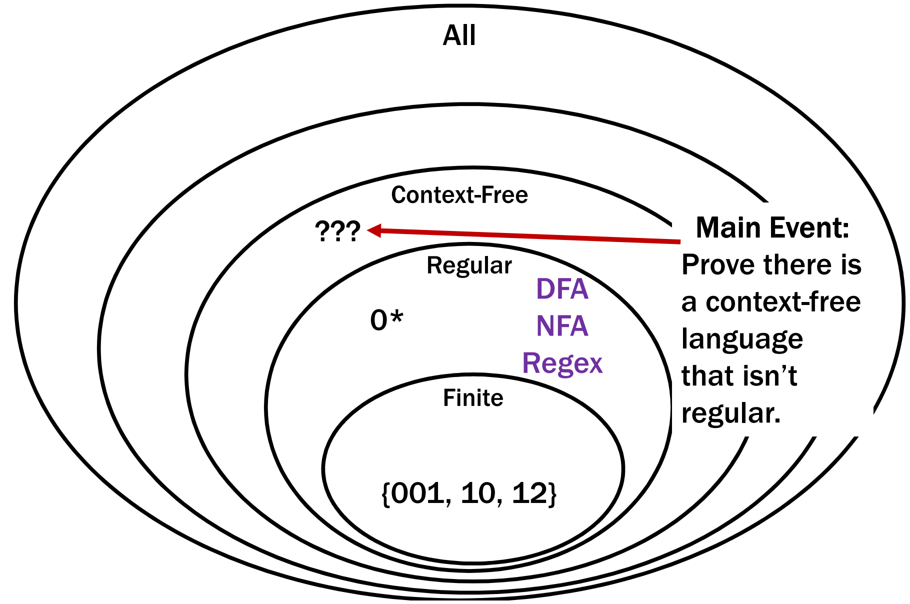
(01 ∪1)*0



NFA to DFA: subset construction



Languages and Machines!



B = {binary palindromes} can't be recognized by any DFA

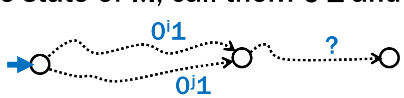
Consider the infinite set of strings

$$S=\{1, 01, 001, 0001, 00001, \ldots\} = \{0^n1 : n \geq 0\}$$

Suppose we are given an arbitrary DFA M.

- Goal: Show that some $x \in B$ and some $y \notin B$ both must end up at the *same* state of M

Since S is infinite we know that two different strings in S must land in the same state of M, call them 0^i1 and 0^j1 for $i \neq j$.



- That also must be true for 0^i1z and 0^j1z for any $z \in \{0,1\}^*$!

In particular, with $z=0^i$ we get that 0^i10^i and 0^j10^i end up at the same state of M. Since $0^i10^i \in B$ and $0^j10^i \notin B$ (because $i \neq j$) M does not recognize B. ∴ no DFA can recognize B.

Showing a Language L is not regular

1. Find an infinite set $S=\{s_0,s_1,\ldots,s_n,\ldots\}$ of string prefixes that you think will need to be remembered separately
2. "Let M be an arbitrary DFA. Since S is infinite and M is finite state there must be two strings s_i and s_j in S for some $i \neq j$ that end up at the same state of M."

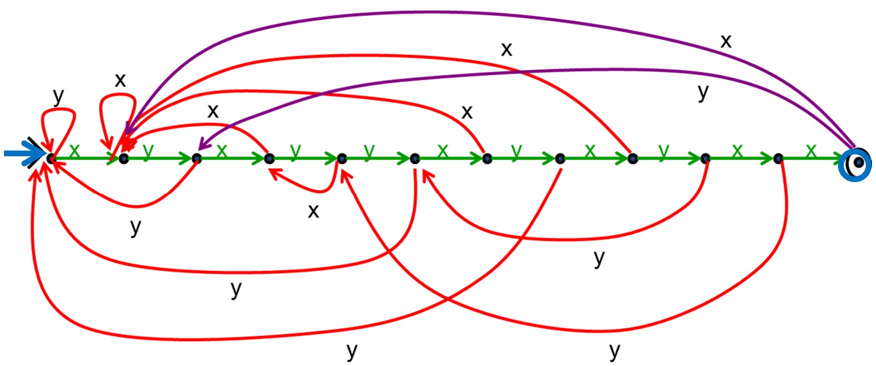
   Note: You don't get to choose which two strings s_i and s_j
3. Find a string t (typically depending on s_i and/or s_j) such that

   s_it is in L, and s_jt is not in L

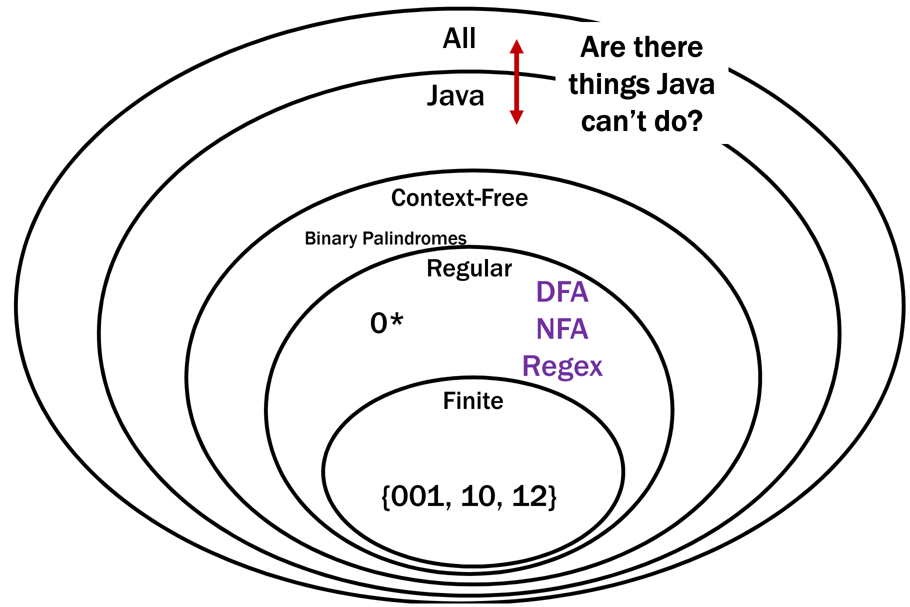
   or s_it is not in L, and s_jt is in L
4. "Since s_i and s_j both end up at the same state of M, and we appended the same string t, both s_it and s_jt end at the same state of M. Since $s_it \in L$ and $s_jt \notin L$, M does not recognize L."
5. "Since M was arbitrary, no DFA recognizes L."

Pattern Matching DFA

pattern **p**=x y x y y x y x y x x

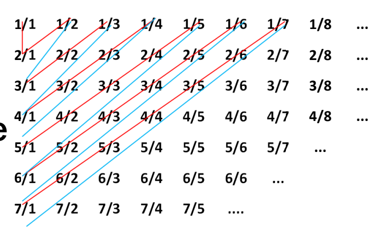


Languages and Machines!



cardinality

- A set S is *countable* iff we can write it as $S=\{s_1, s_2, s_3, \ldots\}$ indexed by $\mathbb{N}$
- Set of integers is countable
  - {0, 1, -1, 2, -2, 3, -3, 4, . . .}
- Set of rationals is countable
  - "dovetailing"



- Σ^* is countable
  - {0,1}* = {0,1,00,01,10,11,000,001,010,011,100,101,...}
- Set of all (Java) programs is countable

Flipped Diagonal Number D

		1	2	3	4						
D=	0.	1									
r_1	0.	5^{1}	0	0	0						
r_2	0.	3	3^{5}	3	3						
r_3	0.	1	4	2^{5}	8	5	7	1	4	...	...
r_4	0.	1	4	1	5^{1}	9	2	6	5	...	...
r_5	0.	1	2	1	2	2^{5}	1	2	2	...	...
r_6	0.	2	5	0	0	0	0^{5}	0	0	...	...
r_7	0.	7	1	8	2	8	1	8^{5}	2	...	...
r_8	0.	6	1	8	0	3	3	9	4^{5}	...	...
...	....	...	....	....	...	...	...	...	...	......	

Flipping Rule:

If digit is 5, make it 1
If digit is not 5, make it 5

The Halting Problem

Given: - CODE(**P**) for any program **P**

- input **x**

Output: **true** if **P** halts on input **x**

false if **P** does not halt on input **x**

Theorem (Turing): There is no program that solves the halting problem

"The halting problem is undecidable"

Does **D**(CODE(**D**)) halt?

```
public static void D(x) {
    if (H(x,x) == true) {
        while (true); /* don't halt */
    }
    else {
        return;       /*    halt    */
    }
}
```

H solves the halting problem implies that
H(CODE(**D**),x) is **true** iff **D**(x) halts, H(CODE(**D**),x) is **false** iff not

Suppose **D**(CODE(**D**)) **halts**.
Then, we must be in the **second** case of the if.
So, H(CODE(**D**), CODE(**D**)) is **false**
Which means **D**(CODE(**D**)) **doesn't halt**

Suppose **D**(CODE(**D**)) **doesn't halt**.
Then, we must be in the **first** case of the if.
So, H(CODE(**D**), CODE(**D**)) is **true**.
Which means **D**(CODE(**D**)) **halts**.

Contradiction!

Some possible inputs **x**

D behaves like **flipped diagonal**

programs **P**

	$\langle P_1\rangle$	$\langle P_2\rangle$	$\langle P_3\rangle$	$\langle P_4\rangle$	$\langle P_5\rangle$	$\langle P_6\rangle$	....					
P_1	0 1	1	1	0	1	1	1	0	0	0	1	...
P_2	1	1 0	0	1	0	1	1	0	1	1	1	...
P_3	1	0	1 0	0	0	0	0	0	0	0	1	...
P_4	0	1	1	0 1	1	0	1	1	0	1	0	...
P_5	0	1	1	1	1 0	1	1	0	0	0	1	...
P_6	1	1	0	0	0	1 0	1	0	1	1	1	...
P_7	1	0	1	1	0	0	0 1	0	0	0	1	...
P_8	0	1	1	1	1	0	1	1 0	0	1	0	...
P_9	.	.	.	.	.	.	.	.	.	.	.	
.	.	.	.	.	.	.	.	.	.	.	.	
.												

(**P**,**x**) entry is **1** if program **P** halts on input **x** and **0** if it runs forever

---

But first another hard halting-related problem

Halting Problem:

Given: - CODE(**P**) for any program **P**

- input **x**

Output: **true** if **P** halts on input **x**

false if **P** does not halt on input **x**

HaltsNoInput Problem:

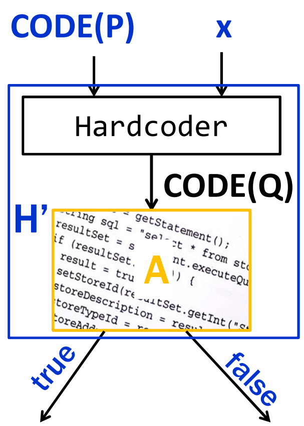
Given: - CODE(**Q**) for any program **Q**

Output: **true** if **Q** halts without reading any input

false if **Q** reads input or runs forever without reading any input.

Showing there is no program solving HaltsNoInput

Suppose that hypothetical program A solves **HaltsNoInput** problem. *Combine with* `Hardcoder`:



H' *outputs true on inputs*
  CODE(P) *and* x
iff A outputs true
  on input CODE(Q) by diagram
iff Q() reads no input
  and (always) halts by property of A
iff P(x) *halts* by definition of Hardcoder

If A existed then H' would solve the Halting Problem: Impossible

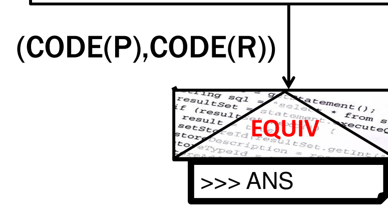
Showing EQUIV is Undecidable

Consider the set:
EQUIV = {(CODE(P), CODE(R)): P, R are programs, P(x) = R(x) for all inputs x}

CODE(Q)

Question: Does Q() halt?

Step 1: Construct P:
```
public static boolean P() {return true;}
```
Step 2: Construct R:
- Step a: Replace return type of Q with boolean
- Step b: Replace all return values with true
- Step c: Add "return true;" to the end of the program
- Call this program R



Combined program outputs **true**
iff **P** and **R** are equivalent by diagram
iff **R** always returns **true** by defn of P
iff **Q** halts by construction of **R** from **Q**

Turing machines

Church-Turing Thesis

Any reasonable model of computation that includes all possible algorithms is equivalent in power to a Turing machine

- Evidence
  - Intuitive justification
  - Huge numbers of equivalent models to TM's based on radically different ideas

what is a Turing machine?

what is a turing machine?

sample Turing machine

	_	0	1
s_1	$(1,s_3)$	$(1,s_2)$	$(0,s_2)$
s_2	(H,s_3)	(R,s_1)	(R,s_1)
s_3	(H,s_3)	(R,s_3)	(R,s_3)

_	_	1	1	0	1	1	_	_

Turing's big idea: machines as data

- **Original Turing machine definition**
  - **A different "machine" M for each task**
  - **Each machine M is defined by a finite set of possible operations on finite set of symbols**
    - M has a finite description as a sequence of symbols, its "code" denoted <M>
- **You already are used to this idea with the notion of the program code or text but this was a new idea in Turing's time.**

Turing's idea: a Universal Turing Machine

- **A Turing machine interpreter U**
  - **On input <M> and its input x, U outputs the same thing as M does on input x**
  - **At each step it decodes which operation M would have performed and simulates it.**
- **One Turing machine is enough**
  - **Basis for modern stored-program computer**
    - Von Neumann studied Turing's UTM design

input x → M → output M(x)

x, <M> → U → output M(x)

General phenomenon: can't tell a book by its cover

and you can't tell what a program does just by its code…

Rice's Theorem: In general there is no way to tell anything about the input/output (I/O) behavior of a program P just given it code!

Note: The statement above is not precise, and we didn't prove it, so this isn't something you can use on homework or exams

Quick lessons

- Don't rely on the idea of improved compilers and programming languages to eliminate major programming errors
  - truly safe languages can't possibly do general computation
- Document your code!!!!
  - there is no way you can expect someone else to figure out what your program does with just your code ….since….in general it is provably impossible to do this!