Computability

Robert M. Keller
Harvey Mudd College
28 March 2005

What is Computability?

- Computability is the study of what kinds of things can be computed, and the kinds of machines needed to compute them.

What is a Computing Machine?

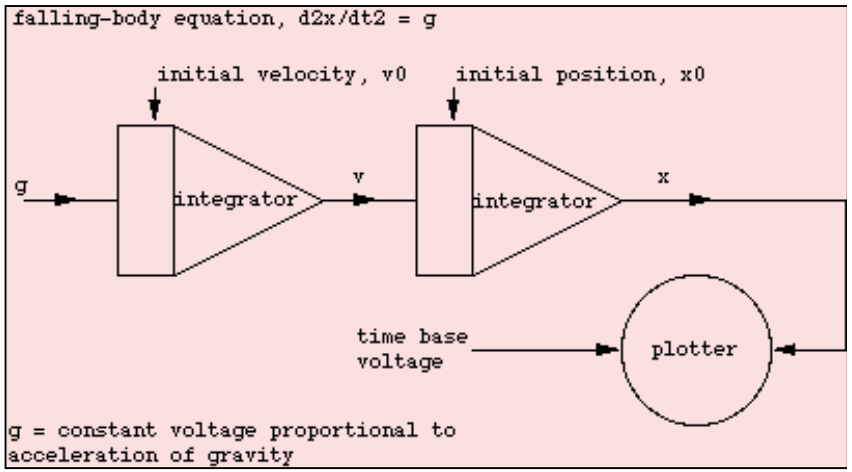
- Computability studies computing in terms of abstract mathematical models, called “machines”.

- These models are idealized versions of physically-realizable computers and their software.

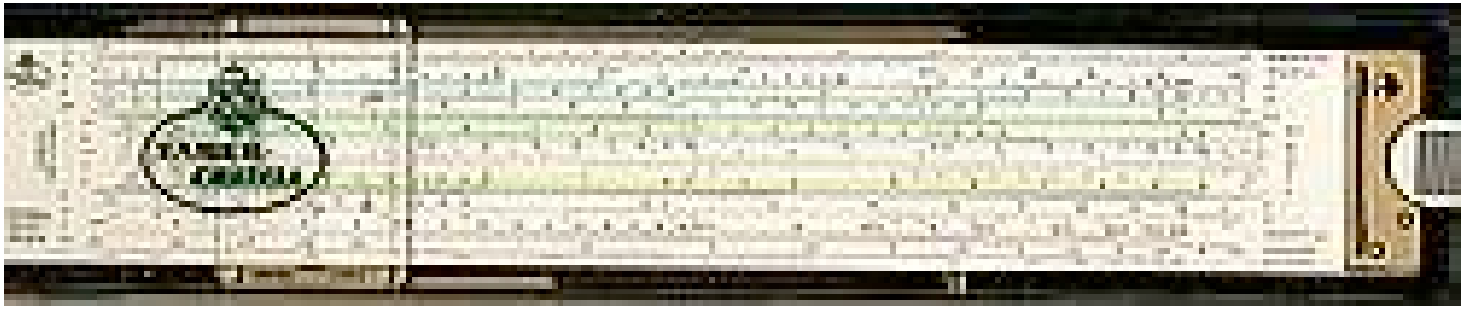
Requirements for Computing

- We deal with *discrete* computation only:
  - Computation is accomplished in discrete, identifiable, steps.
  - Data can be mapped into strings or natural numbers.
  - Don’t deal directly with “real numbers”, but rather (arbitrarily-close) approximations to them.
  - [It isn’t clear that real numbers are “real” anyway. They *are* a useful mathematical artifice.]

Analog Computing



- Analog measures, digital counts.
- We don't treat analog directly.
- Continuous-time
- Ostensibly real-numbers.
- However, "numbers" less accurate than digital computing.
- Can be simulated by digital computing



Preliminary Concepts

- Sets: assumed
- Natural numbers:
  - Can be characterized in the abstract by axioms.
  - Model can be “constructed” out of sets alone.

Numbers from Sets

- Identify 0 as the empty set {} also shown as ∅.
- For an number (set) n, the successor s(n) is definable by:

$$s(n) = \{n\} \cup n$$

- $s(0) = \{0\} \cup 0 = \{0\} \cup \{\} = \{0\}$
- $s(s(0)) = \{s(0)\} \cup s(0) = \{\{0\}\} \cup \{0\} = \{s(0), 0\}$
- $s(s(s(0))) = \{s(s(0))\} \cup s(s(0)) = \{s(s(0)), s(0), 0\}$ etc.
  Note that $s^n(0)$, the n-fold application of s to 0, has "n" elements.

The Natural Numbers N

- N = {0, s(0), s(s(0)), s(s(s(0))), …}
  also shown as {0, 1, 2, 3, …}

- N is the smallest set such that:
  - $0 \in N$
  - If $n \in N$ then $s(n) \in N$.

Functions and Partial Functions

- A binary predicate p on sets A and B is a **function** from A to B provided:
  1. (totality)
     $$\forall x \in A \;\; \exists y \in B \;\; p(x, y)$$
     and
  2. (uniqueness)
     $$\forall x \in A \;\; \forall y \in B \;\; \forall y' \in B$$
     $$((p(x, y) \wedge p(x, y')) \rightarrow y = y')$$

- If condition 2 holds (but 1 may or may not) we say the predicate is a **partial function** from A to B.
- Note:
  - $\forall x \in A \; \varphi$ is an abbreviation for $\forall x ((x \in A) \rightarrow \varphi)$
  - $\exists x \in A \; \varphi$ is an abbreviation for $\exists x ((x \in A) \wedge \varphi)$
  - Thus $\neg(\forall x \in A \; \varphi)$ equates to $\exists x \in A \; \neg\varphi$, etc.

Total vs. Partial Functions

- "Partial" is not an adjective modifying function, even though it appears as such.
- "Partial Function" is the general concept, and "Function" is a special case.
- The term "Total Function" may be used to emphasize the case when a partial function happens to be a function too.

Properties of Functions

- A function is **onto** provided:
  - $\forall y \in B\ \exists x \in A\ p(x, y)$
    (which is totality "flipped")
- A function is **one-to-one** (1-1) provided:
  - $\forall y \in B\ \forall x \in A\ \forall x' \in A\ (p(x, y) \wedge p(x', y)) \rightarrow x = x'$
    (which is uniqueness flipped)
- Onto functions are also called **surjections** (French: "sur" = "on").
- One-to-one functions are also called **injections.** (French: "in" = ???).
- Functions that are both are called **bijections**.

1-1 Correspondence

- A bijection is also called a **1-1 correspondence**.

- There is a 1-1 correspondence between two sets iff their elements can be **exactly paired up**, with no leftovers on either side.

Examples

- There is a 1-1 correspondence between the sets in each of these pairs:
  - {0, 1, 2} — {a, b, c}
  - {0, 1, 2, …} — {1, 2, 3, …}
  - {0, 1, 2, …} — {2, 4, 6, …}
  - {0, 1, 2, …} — {1, 3, 5, …}
  - {0, 1, 2, …} — {1, 2, 3, 5, 7, 11, …}
  - {0, 1, 2, …} — {Λ, a, b, aa, ab, ba, bb, …} (strings)
  - set of all tautologies* — set of all unsatisfiable formulas
  - set of all tautologies* — set of all satisfiable formulas
  - set of all tautologies* — {0, 1, 2, …}

  * assuming a finite set of proposition symbols

Using Function Properties to Compare Set "Sizes"

- $|A| \leq |B|$ means
  there is an injection from A to B.

- $|A| = |B|$ means
  $|A| \leq |B|$ and $|B| \leq |A|$.

- $|A| < |B|$ means
  $|A| \leq |B|$ and not $|A| = |B|$.

Schroeder-Bernstein Theorem

- If there is an injection from A to B and an injection from B to A, then there is a bijection from A to B.

- Makes intuitive sense, but is there an *obvious* proof?

Countability and Finiteness

- A set S is called **countable** iff there is a 1-1 correspondence between it and a subset of the natural numbers N, i.e. $|S| \leq |N|$.

- A set is called **finite** iff there is a 1-1 correspondence between it and *some* natural number (i.e. one of 0, s(0), s(s(0)), …) taken as a set.

- A set is called **infinite** if it is not finite, and **countably-infinite** if it is both infinite and countable.

Examples of Countable Sets

- Finite:
  - {} {0, 1, 2} {a, b, c, …., z}
- Countably-Infinite:
  - {0, 1, 2, …} {1, 2, 3, …}
  - {0, 1, 2, …} {2, 4, 6, …}
  - {0, 1, 2, …} {1, 3, 5, …}
  - {0, 1, 2, …} {1, 2, 3, 5, 7, 11, …} (primes)
  - all strings over a finite alphabet, e.g. {Λ, a, b, aa, ab, ba, bb, …}
  - set of all tautologies* set of all unsatisfiable formulas
  - set of all tautologies* set of all satisfiable formulas
  - set of all tautologies* {0, 1, 2, …}

* assuming a finite set of proposition symbols

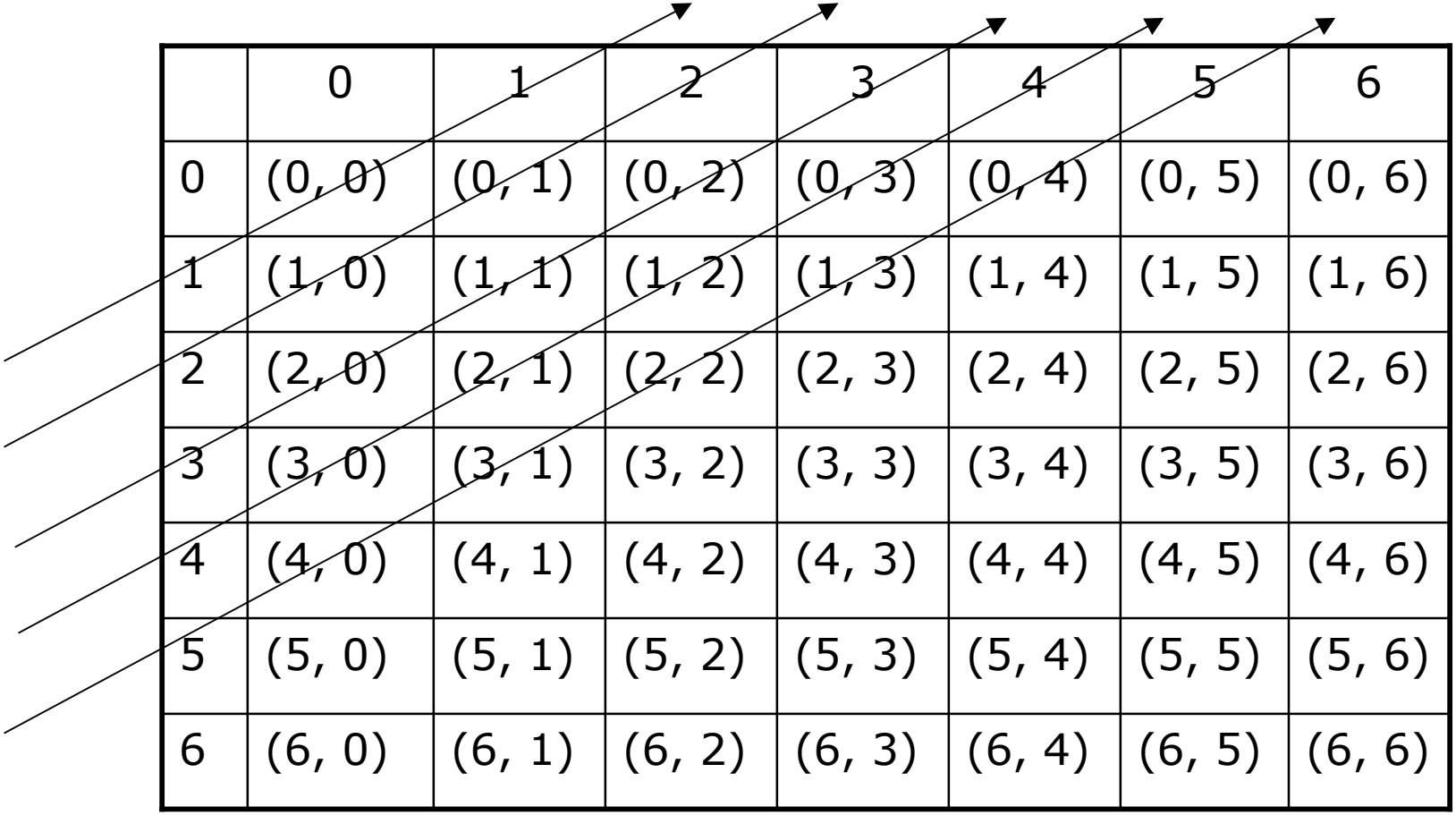
More Countable Sets

- The set of all pairs of natural numbers:
  (0, 0), (1, 0), (0, 1), (2, 0), (1, 1), (0, 2), …

- The set of all finite sequences of natural numbers:
  (),
  (1), (0),
  (2), (0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2),
  (3), (0, 3), (1, 3), (2, 3), (3, 0), (3, 1), (3, 2), (3, 3), (0, 0, 0),
  (0, 0, 1), (0, 0, 2), (0, 0, 3), (0, 1, 0), (0, 1, 1), (0, 1, 2), (0, 1, 3),
  (0, 2, 0), (0, 2, 1), (0, 2, 2), (0, 2, 3), (0, 3, 0), (0, 3, 1), (0, 3, 2),
  (0, 3, 3), (3, 0, 0), (3, 0, 1), (3, 0, 2), (3, 0, 3), (3, 1, 0), (3, 1, 1),
  (3, 1, 2), (3, 1, 3), (3, 2, 0), (3, 2, 1), (3, 2, 2), (3, 2, 3), (3, 3, 0),
  (3, 3, 1), (3, 3, 2), (3, 3, 3),
  (4), …

“Stripe” Diagram for All Pairs

sum to: 0 1 2 3 4 …



	0	1	2	3	4	5	6
0	(0, 0)	(0, 1)	(0, 2)	(0, 3)	(0, 4)	(0, 5)	(0, 6)
1	(1, 0)	(1, 1)	(1, 2)	(1, 3)	(1, 4)	(1, 5)	(1, 6)
2	(2, 0)	(2, 1)	(2, 2)	(2, 3)	(2, 4)	(2, 5)	(2, 6)
3	(3, 0)	(3, 1)	(3, 2)	(3, 3)	(3, 4)	(3, 5)	(3, 6)
4	(4, 0)	(4, 1)	(4, 2)	(4, 3)	(4, 4)	(4, 5)	(4, 6)
5	(5, 0)	(5, 1)	(5, 2)	(5, 3)	(5, 4)	(5, 5)	(5, 6)
6	(6, 0)	(6, 1)	(6, 2)	(6, 3)	(6, 4)	(6, 5)	(6, 6)

Note

- Simply putting … at the end of a partial sequence does not make the set countable.
- One must be able to describe a pattern that **enumerates** the set ***without omissions*** (duplicates are ok).
- That is, explicitly exhibit a function f such that {f(0), f(1), f(2), …} is **onto** the set in question.

Examples of Enumeration

- f(n) = 2n enumerates the set of even numbers
- f(n) = 2n+1 enumerates the set of all odd numbers
- f(n) = the nth prime number
  enumerates the set of all prime numbers
  (can be done by giving a program)

```
nthPrime(0) => 1;
nthPrime(n+1) => nextPrime(nthPrime(n)+1);

nextPrime(p) = isPrime(p) ? p : nextPrime(p+1);

isPrime(p) = noDivisors(p, 2);

noDivisors(p, n) = n == p ? 1 : divides(n, p) ? 0 : noDivisors(p, n+1);
```

Alternatives to Enumeration

- Any **subset** of a countable set is countable.

- Any set in **1-1 correspondence** with a known countable set is countable.

Example

- The set of all **finite subsets** of natural numbers is countable.

- (Why?)

An *Un*countable Set

- The set of **all subsets** the natural numbers is ***not*** countable.

- This theorem was first proved by Georg Cantor (1845-1918) in 1873.

Proof of Cantor's Theorem

- Suppose the set of all subsets of natural numbers can be enumerated.
- Let $f:N \rightarrow$ all subsets of N be the enumerating function.
- Define a set

$$S = \{n \in N \mid n \notin f(n)\}$$

Proof of Cantor's Theorem

- $S = \{n \in N \mid n \notin f(n)\}$
- Obviously $S \subseteq N$. Therefore, there is a $k \in N$ such that **$f(k) = S$**.

- In particular, $k \in S$ iff $k \notin f(k)$.

- But $f(k)$ ***is*** S, so $k \in S$ iff $k \notin S$.

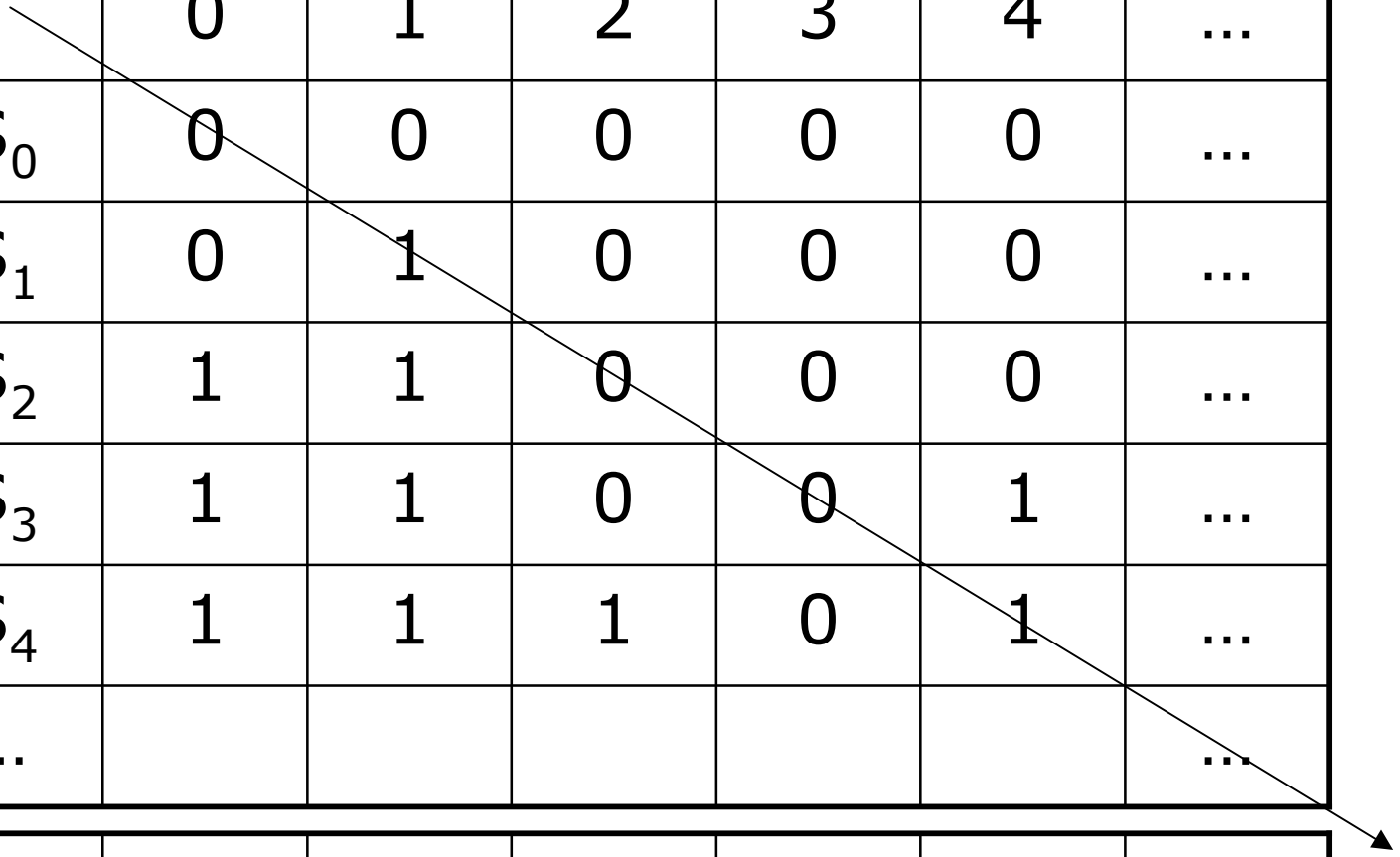
- By reductio ad absurdum, the supposed function f does not exist.

Diagonalization Principle

- The elements of N are columns, and the supposed subsets of N are rows, of an infinite matrix.
- Put a 1 in row i column j if i $\in$ j^{th} subset.
- Create a row by flipping the diagonal elements, 1 for 0, 0 for 1.
- This row is guaranteed not to be a row of the original matrix.

Diagonalization Principle

	0	1	2	3	4	…
S_0	0	0	0	0	0	…
S_1	0	1	0	0	0	…
S_2	1	1	0	0	0	…
S_3	1	1	0	0	1	…
S_4	1	1	1	0	1	…
…						…
new	1	0	1	1	0	



Ways of Showing a Set **Un**countable

- Diagonalization (as on previous slide)
- Showing that the set in question has a known uncountable set as a subset.
- Showing there is a mapping from the set in question **onto** a known uncountable set.

Example

- The set of all functions of the form
  $N \rightarrow \{0, 1\}$
  is uncountable.

- Proof: The set in question is in 1-1 correspondence with the set of all subsets of N:
  - For each $S \subseteq N$, define $f_S(j) = 1$ if $j \in S$, 0 otherwise.
  - For each function $f: N \rightarrow \{0, 1\}$, define $S_f = \{j \in N \mid f(j) = 1\}$.

- f_S is called the *characteristic function* of S.

Requirements for Computability (Turing)

- The amount of memory occupied at any one time is finite.
- (The amount of memory used over all time is countable, but not necessarily finite.)
- The set of symbols out of which information is constructed is finite.
- The “set of instructions” or “program” for directing the computation is composed of a finite set of rules from those symbols.
- The applicability and action of a rule can be determined by examining a finite amount of information.

Examples of Computability (Low end to high-end)

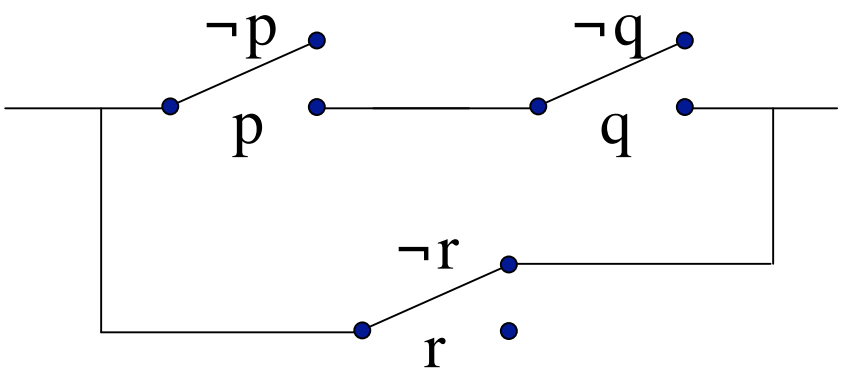
- Switching circuits
- Finite state machines
- Pushdown automata
- Linear-bounded automata
- Turing machines
- Partial recursive functions
- Markov algorithms
- Post machines
- Lambda calculus

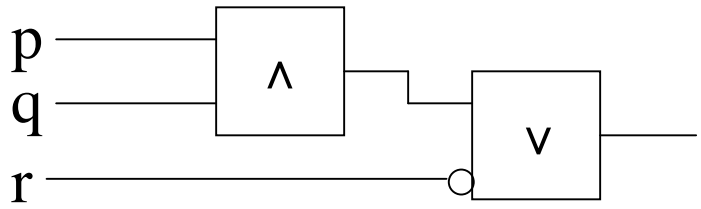
. . .

equivalent

Switching Circuits

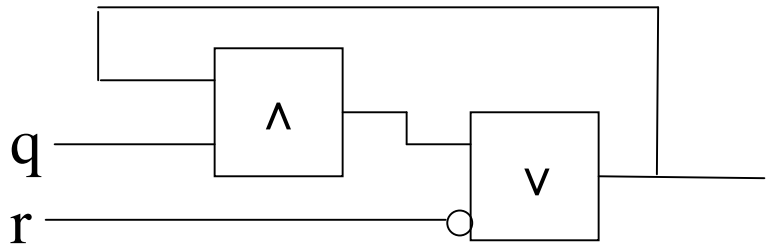
- Relays
- Gates

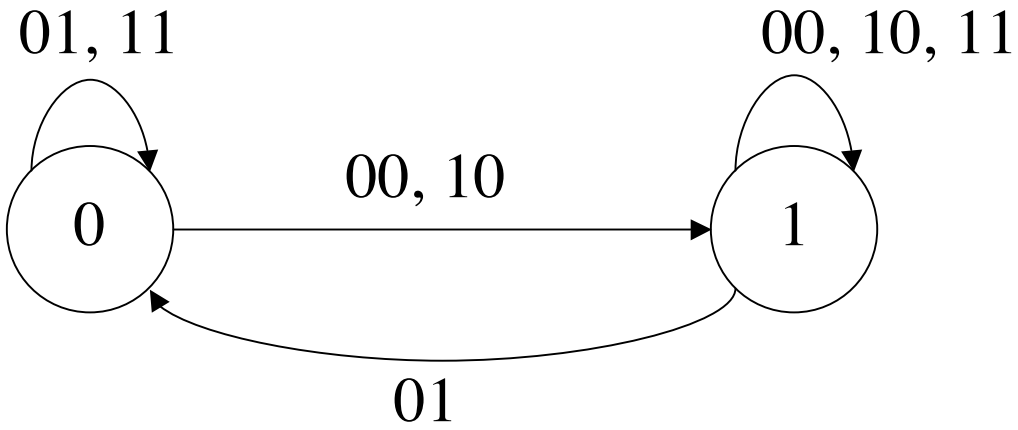




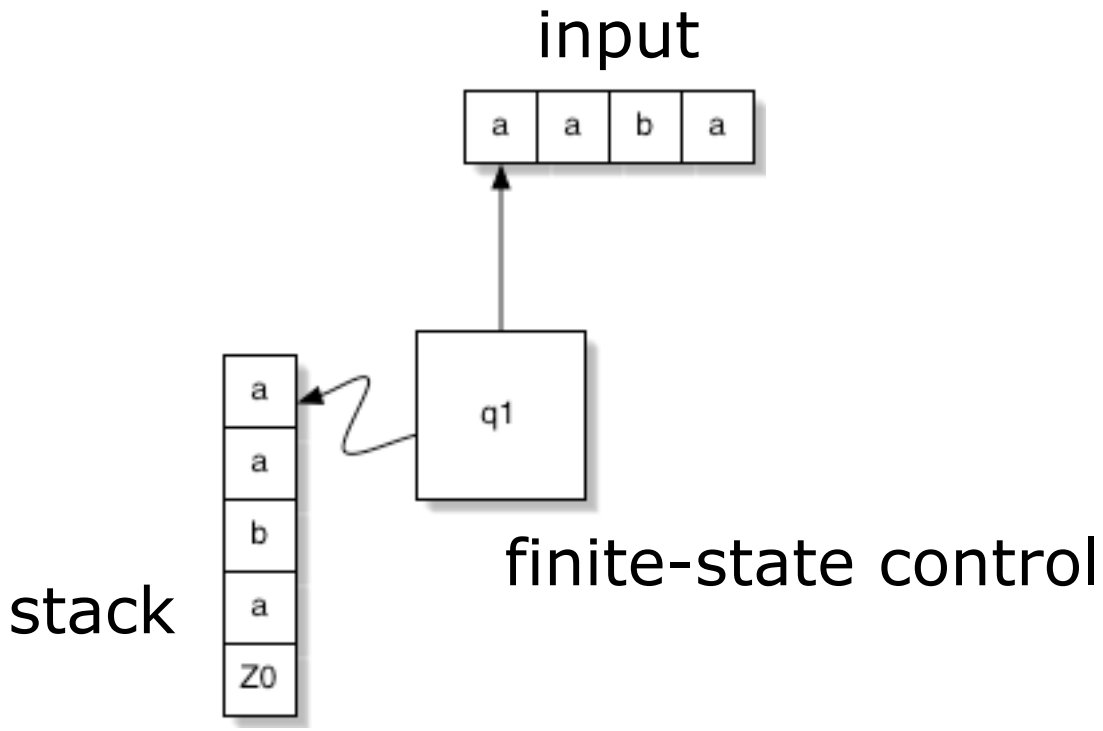
transmission = $(p \wedge q) \vee \neg r$

Finite-State Machines





Pushdown Automaton (PDA)
(used for studying language parsing)



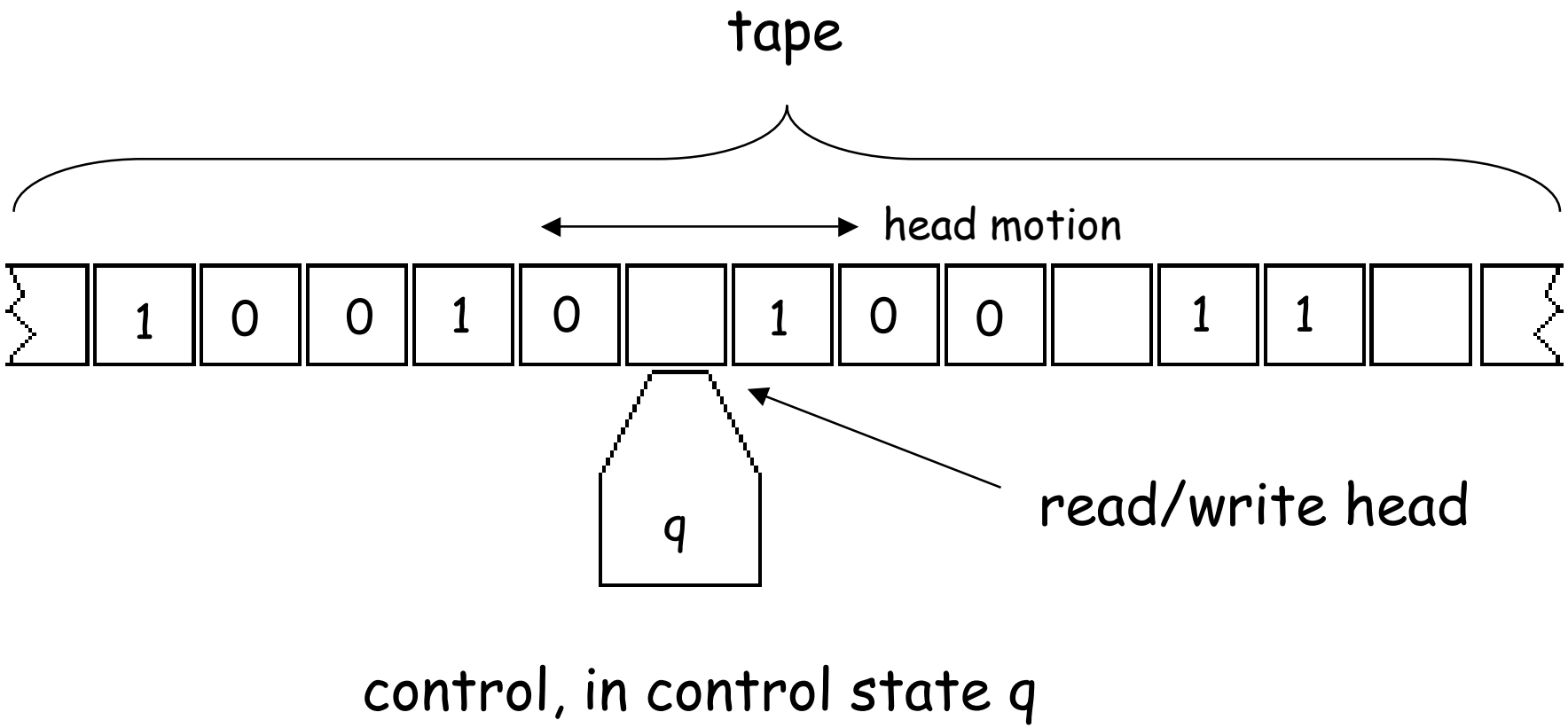
Turing Machine

- Named after (not by) Alan M. Turing.
- Perhaps the most important computational model, from a theoretical viewpoint.
- Simplistic, yet universal.

Relation to Previous Machines

- Like a FSM but
  - can **write** as well as read its tape
  - can move in **both directions**
- Like a PDA with *two* stacks
- More powerful than a locomotive, able to …

Turing Machine Diagram



Turing Machine Components

- $(Q, \Sigma, \Gamma, \delta, q_0, B, A)$
- Q: finite set of control states
- Σ: input alphabet
- Γ: tape alphabet ($\Sigma \subset \Gamma$)
- δ: transition function (next page)
- q_0: initial control state
- B: the blank symbol $B \in (\Gamma - \Sigma)$
- A: accepting control states ($A \subseteq Q$)

Turing Machine Components

- δ: transition partial function

  $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times M$

- where M = set of possible head motions
  - R = right
  - L = left
  - N = none

TM Transition Function

- For each $q \in Q$ and $X \in \Gamma$,

  $\delta(q, X) = (q', Y, D)$ or is unspecified

  where $q' \in Q$, $Y \in \Gamma$, and $D \in \{R, L, N\}$

- means
  - the machine is in state q and reading X
  - the machine writes Y,
    the control goes to state q',
    and the head moves in direction D

TM Rule Notation

- Instead of

  $\delta(q, X) = (q', Y, D)$

  we may choose to write

  $q, X \rightarrow q', Y, D$

  or just

  (q, X, q', Y, D)

- This is called "5-tuple" notation.
- We need to make sure that δ thus specified is a partial function.
- If **no action is specified** for a given combination q, X, the convention is that the machine stops.

Standardized Input Assumption

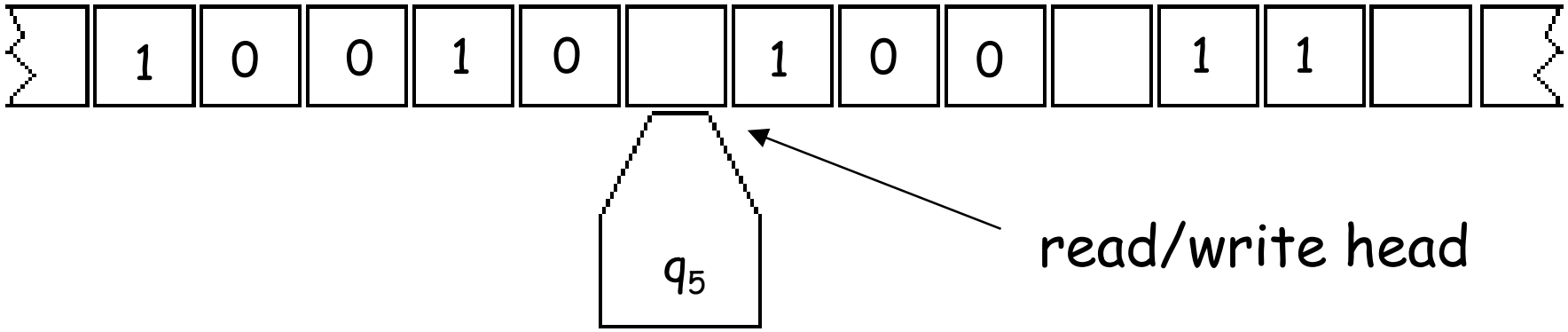
- The input is a finite string containing only symbols in the input alphabet Σ.
- In particular, there are no blanks (B) embedded in the input, since B $\in (\Gamma\text{-}\Sigma)$.
- The tape outside of the input initially contains only blanks.

TM Configuration (i.e. complete state)

- A TM configuration consists of: (L, S, H, R), where
  - L = the part of the tape to the left of the read-write head, including at least all non-blank symbols on the left.
  - S = a control state.
  - H = the symbol under the read-write head.
  - R = the part of the tape from the right of the read-write head onward, including at least all non-blank symbols on the right.

Example Configuration

(L, S, H, R) = (10010, q_5, _, 100_11)
where _ represents the blank symbol



General Transitions

- A machine makes one of the following transitions:
  - $(w, q, X, Yz) \vdash (wX', q', Y, z)$ if $\delta(q, X) = (q', X', \mathbf{R})$
  - $(wX, q, Y, z) \vdash (w, q', X, Y' z)$ if $\delta(q, X) = (q', X', \mathbf{L})$
  - $(w, q, X, z) \vdash (w, q', X', z)$ if $\delta(q, X) = (q', X', \mathbf{N})$
- If no transition is specified, the machine stops.

Our Standard Starting Configuration C_0

- The machine starts in q_0 .
- The head is positioned over a blank just to the left of the first input symbol (if any).

$$(\Lambda, q_0, b, x) \text{ where } x \in \Sigma^*$$

Σ^* is the set of all finite strings of symbols in Σ.

Λ is the empty string.

Machine Behavior

- Two general behaviors are possible:
  - Halting behavior: $C_0 \vdash C_1 \vdash C_2 \vdash \ldots \vdash C_n$
  - Diverging behavior: $C_0 \vdash C_1 \vdash C_2 \vdash \ldots$ (non-terminating)
- where C_0 is the starting configuration
- and C_n is a configuration from which no transition is specified (called a **halting configuration**).

Acceptance

- The original input is **accepted** if the machine reaches a halting configuration for which the control state is accepting.
- The original input is **explicitly rejected** if the machine reaches a halting configuration for which the control state is not accepting.
- The original input is **implicitly rejected** if the machine never reaches a halting configuration.
- We use **rejected** to represented either of the latter two.

Rules for a TM accepting $\{a^n b^n c^n \mid n \in N\}$

$Q = \{q_0, q_a, q_b, q_c, q_A, q_L, q_R, h_a, h_r\}$, $\Gamma = \{a, b, c, \Delta, x\}$, $A = \{h_a\}$

- $q_0, b \rightarrow q_a, \Delta, R$
- $q_a, \Delta \rightarrow q_A, \Delta, L$
- $q_a, x \rightarrow q_a, x, R$
- $q_a, a \rightarrow q_b, x, R$
- $q_b, b \rightarrow q_c, x, R$
- $q_b, x \rightarrow q_b, x, R$
- $q_c, c \rightarrow q_L, x, L$
- $q_L, c \rightarrow q_L, c, L$
- $q_L, b \rightarrow q_L, b, L$
- $q_L, a \rightarrow q_L, a, L$
- $q_L, x \rightarrow q_L, x, L$
- $q_L, \Delta \rightarrow q_a, \Delta, R$
- $q_A, x \rightarrow q_A, x, L$
- $q_A, \Delta \rightarrow h_a, \Delta, S$
- $q_a, b \rightarrow q_R, b, L$
- $q_a, c \rightarrow q_R, c, L$
- $q_b, a \rightarrow q_b, a, R$
- $q_b, c \rightarrow q_R, c, L$
- $q_c, b \rightarrow q_c, b, R$
- $q_c, a \rightarrow q_R, c, L$
- $q_c, x \rightarrow q_c, x, R$
- $q_b, \Delta \rightarrow q_R, \Delta, L$
- $q_c, \Delta \rightarrow q_R, \Delta, L$
- $q_R, \Delta \rightarrow h_R, \Delta, S$
- $q_R, a \rightarrow q_R, a, L$
- $q_R, b \rightarrow q_R, b, L$
- $q_R, c \rightarrow q_R, c, L$
- Any other combinations go to q_R with no change in symbol or direction.

Legend for this TM

q_0: Just Starting

q_a: Looking for a, reject anything else.

q_b: Looking for b, skipping over a's.

q_c: Looking for c, skipping over b's, reject anything else.

q_L: Moving left after matching one a, b, and c, looking for blank.

q_R: Moving left after deciding rejection.

q_A: Moving left after deciding acceptance.

h_a: Halt and accept.

h_r: Halt and reject.

A rex Rendering (see /cs/cs81/rex/abc.rex)

```
f("q0", " ") => ["qa", " ", "R"];
f("qa", " ") => ["qA", " ", "L"];
f("qa", "a") => ["qb", "x", "R"];
f("qa", "b") => ["qR", "b", "L"];
f("qa", "c") => ["qR", "c", "L"];
f("qa", "x") => ["qa", "x", "R"];
f("qb", " ") => ["qR", " ", "L"];
f("qb", "a") => ["qb", "a", "R"];
f("qb", "b") => ["qc", "x", "R"];
f("qb", "c") => ["qR", "c", "L"];
f("qb", "x") => ["qb", "x", "R"];
f("qc", " ") => ["qR", " ", "L"];
f("qc", "a") => ["qR", "c", "L"];
f("qc", "b") => ["qc", "b", "R"];
f("qc", "c") => ["qL", "x", "L"];
f("qc", "x") => ["qc", "x", "R"];
```

```
f("qA", " ") => ["ha", " ", ”N"];
f("qA", "x") => ["qA", "x", "L"];
f("qL", " ") => ["qa", " ", "R"];
f("qL", "a") => ["qL", "a", "L"];
f("qL", "b") => ["qL", "b", "L"];
f("qL", "c") => ["qL", "c", "L"];
f("qL", "x") => ["qL", "x", "L"];
f("qR", " ") => ["hr", " ", ”N"];
f("qR", "a") => ["qR", "a", "L"];
f("qR", "b") => ["qR", "b", "L"];
f("qR", "c") => ["qR", "c", "L"];
f("qR", "x") => ["qR", "x", "L"];
```

A rex TM Infrastructure

```
// TM single-step function

step([L, Q, X, M])  => step1(L, f(Q, X), X, M);

// This auxiliary step function discriminates based on the "move" symbol.
// The left tape is reversed for convenient access to its rightmost symbol.

step1(L,       [P, Y, "R"], _, [X | M]) => [[Y|L], P, X, M];          // Right

step1(L,       [P, Y, "N"], _, M)       => [L,     P, Y , M];         // None

step1([Z | L], [P, Y, "L"], _, M)       => [L,     P, Z, [Y | M]];    // Left

step1([],      [P, Y, "L"], _, M)       => [[],    P, " ", [Y | M]];  // L end

step1(L,       [P, Y, "R"], _, [])      => [[Y|L], P, " ", []];       // R end

// TM run Function

run([L, "ha", X, M]) => [[L, "ha", X, M]];            // Halt and accept

run([L, "hr", X, M]) => [[L, "hr", X, M]];            // Halt and reject

run(State) => [State | run(step(State))];             // Keep running
```

rex simulation (slightly edited)

```
[], q0,  , [a, a, b, b, c, c]
[ ], qa, a, [a, b, b, c, c]
[ , x], qb, a, [b, b, c, c]
[ , x, a], qb, b, [b, c, c]
[ , x, a, x], qc, b, [c, c]
[ , x, a, x, b], qc, c, [c]
[ , x, a, x], qL, b, [x, c]
[ , x, a], qL, x, [b, x, c]
[ , x], qL, a, [x, b, x, c]
[ ], qL, x, [a, x, b, x, c]
[], qL,  , [x, a, x, b, x, c]
[ ], qa, x, [a, x, b, x, c]
[ , x], qa, a, [x, b, x, c]
[ , x, x], qb, x, [b, x, c]
[ , x, x, x], qb, b, [x, c]
[ , x, x, x, x], qc, x, [c]
[ , x, x, x, x, x], qc, c, []
[ , x, x, x, x], qL, x, [x]
[ , x, x, x], qL, x, [x, x]
[ , x, x], qL, x, [x, x, x]
[ , x], qL, x, [x, x, x, x]
[ ], qL, x, [x, x, x, x, x]
[], qL,  , [x, x, x, x, x, x]
[ ], qa, x, [x, x, x, x, x]
[ , x], qa, x, [x, x, x, x]
[ , x, x], qa, x, [x, x, x]
[ , x, x, x], qa, x, [x, x]
[ , x, x, x, x], qa, x, [x]
[ , x, x, x, x, x], qa, x, []
[ , x, x, x, x, x, x], qa,  , []
[ , x, x, x, x, x], qA, x, [ ]
[ , x, x, x, x], qA, x, [x,  ]
[ , x, x, x], qA, x, [x, x,  ]
[ , x, x], qA, x, [x, x, x,  ]
[ , x], qA, x, [x, x, x, x,  ]
[ ], qA, x, [x, x, x, x, x,  ]
[], qA,  , [x, x, x, x, x, x,  ]
[], ha,  , [x, x, x, x, x, x,  ]
```

Another Possible Rendering

/cs/cs81/tm provides a TM simulator

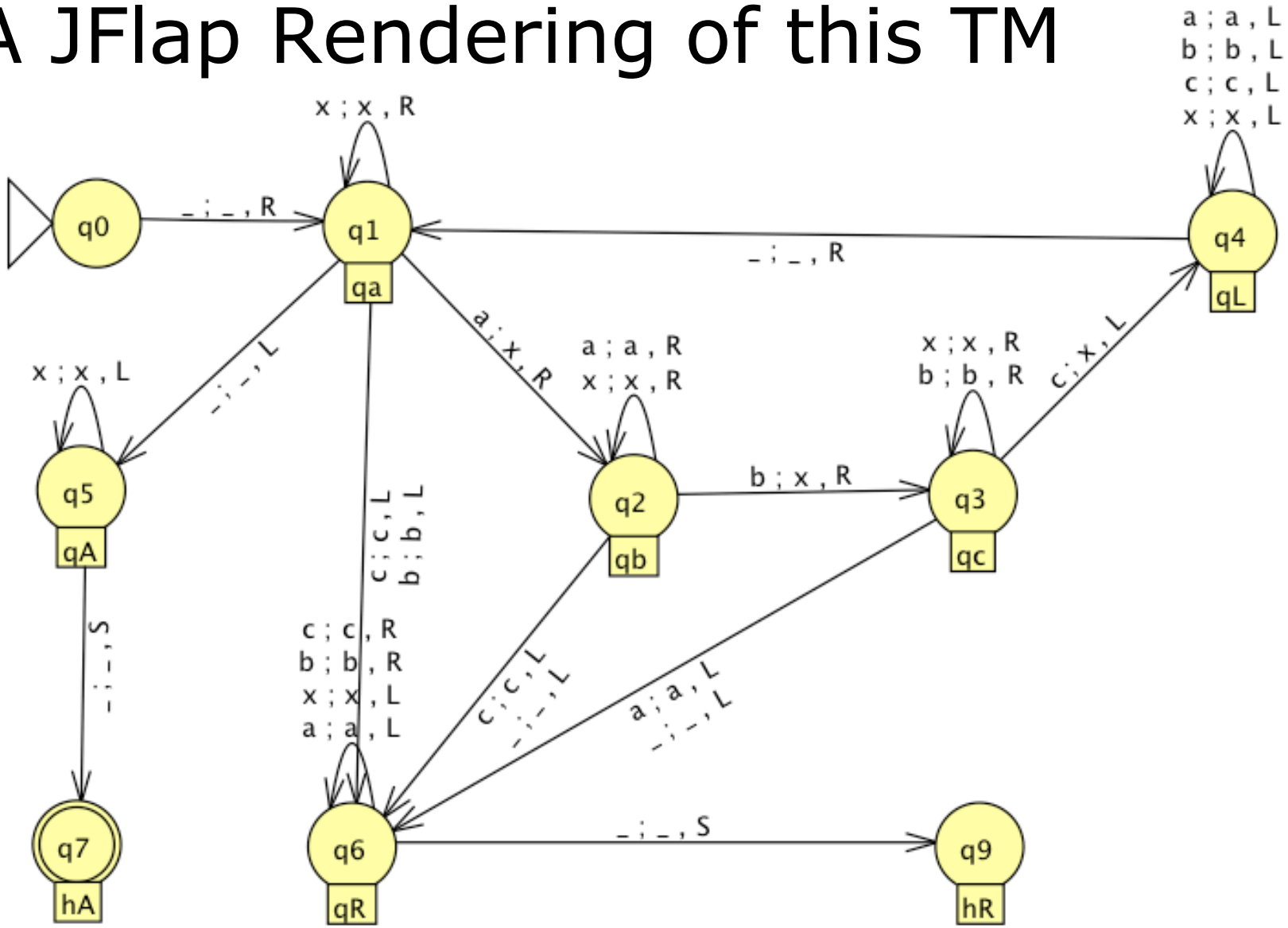
There is documentation (tm.doc) and examples:

add1.tm	Adds 1 to a binary numeral
add.tm	Adds two binary numerals

add1.tm:

start	_	_	left	add1
add1	0	1	right	end
add1	_	1	right	end
add1	1	0	left	add1
end	0	0	right	end
end	1	1	right	end

A JFlap Rendering of this TM



How to use or get JFlap.

- Go to Susan Rodger's (the author's) page:

  www.cs.duke.edu/~rodger/tools/jflap/

- Either:
  - Download the .jar file and run it on your machine, or
  - Use the web applet provided.

- JFlap simulates: FA, NFA, PDA, TM, etc.

Types of Computational Problems

- **Decision**: Determine whether any input string is in a prescribed language or not (i.e. what language is **accepted**).
- **Transformation**: Compute a prescribed function from strings to strings.
- **Enumeration**: With a numeral n as input, generate the nth element of a prescribed set.
- **Optimization**: Find the "best" string satisfying a certain criterion.

Decision ⊂ Transformation

- Language acceptance is equivalent to computing the characteristic function of the language:
- If Σ^* is the universe, and $L \subseteq \Sigma^*$, then

$$c_L(x) = \begin{cases} 1 \text{ if } x \in L \\ \\ 0 \text{ otherwise} \end{cases}$$

- $c_L: \Sigma^* \rightarrow \{0, 1\}$ is the **characteristic function** of L.

Partial Acceptance

- Sometimes we may need to be content if our TM halts in an accepting state when the input is in L, but might not halt at all if it is not.
- We say that the machine **partially accepts** (or **recognizes**) the language L in this more general case.
- As a function, the machine computes a "partial characteristic function":

$$c_L(x) = \begin{cases} 1 \text{ if } x \in L \\ \\ \textit{undefined} \text{ otherwise} \end{cases}$$

TMs as Algorithms

- Obviously every TM is an embodiment of some kind of algorithm.

- So anything computable by a TM could be expressed in an adequate programming language (e.g. rex, as we saw, or many others).

Church/Turing Thesis

- The TM model is *universal*, in the sense that for any computable partial function, there is a Turing machine that computes a suitably-encoded version of that function.

- This goes by other names, such as:
  - Turing's hypothesis
  - Church's thesis
  - etc.

Alonzo Church
1903-1995

Alan Turing
1912-1954

Acceptance of the Church/Turing Thesis

- This hypothesis is nearly universally accepted, for a variety of reasons:
  - Many other proposed universal models have been shown equivalent to TM's by an algorithmic transformation from one model to the other.
  - No one has ever demonstrated a partial function that is compellingly computable, yet could not be also demonstrated to be computable by a TM.

The Church/Turing Hypothesis cannot be proved.

- A proof would correspond to devising another universal model, say M, and showing that TM's are equivalent to M.

- But then we would be in exactly the same position with M as we are with TM's:

  How to prove that M is universal?

Accepting the Church/Turing Thesis can save a lot of work

- For then, we can assert that any reasonably-described algorithm can be implemented on a Turing machine, **without going into all of the details.**

- If details were demanded, they normally be provided beyond a reasonable doubt.

The Church/Turing Thesis is the **foundation** for computability theory.

- In computability theory, broad quantifications over computable functions are performed.

- The impact of such quantifications relies on the universality of the Turing machine model (or any equivalent model) as being able to represent **arbitrary** algorithms.

TM Descriptions as Strings

- As seen earlier, it is possible to give descriptions of Turing machines in the form of strings.
- Consequently, we can talk about **algorithms** that take descriptions of Turing machines as input, and return descriptions of Turing machines as output.
- Moreover, by the Church/Turing thesis, these algorithms can be implemented on Turing machines.

Analogy in Every-Day Computing

- A **compiler** produces an executable from a string representing a program.
- An **interpreter** computes the function specified by a program, its input.
- A **pretty-printer** re-formats a program from a single input string.

Programs that Operate on their Own Representation

- A compiler for a language can usually be written in the language itself.
- Thus the compiler can compile itself.
- This is useful in “bootstrapping” a compiler, beginning with a subset of a language, then adding more features.

Encoding TMs in a Finite Alphabet

- Although there is no limit to the number of symbols we could use to specify Turing machines, we could **encode** all such specifications using a fixed alphabet.
- For example, rather than use state symbols $\{q_0, q_1, q_2, \ldots\}$, we could use the symbols $\{q, 0, 1\}$ and encode the subscripts as binary strings: {q0, q1, q01, q10, q11, q001, …} for example.
- Similarly, we can encode an arbitrary tape alphabet in binary {a0, a1, a01, a10, a11, a001, …}.
- We can go on to encode all rules and all tapes.

The Turing Machine Language

- Given an adequate alphabet for encoding TM descriptions, we can see that:

1. The set of all valid TM descriptions is a **language**.

2. There is a TM that accepts this language.

Notation for Analysis of Turing Machines

- A TM can be defined that performs various analysis tasks on encoded TM's in general.
- If M is a TM in the abstract, then we will use <M> to denote the encoded description of M.
- If M is a TM, and x is an input tape for M, then we will use <M, x> to denote the encoded description of M along with the encoded input tape.

Example of an Analysis Problem

- Consider this partial function:

$$\text{nbp}(\langle M \rangle) = \begin{cases} 1 \text{ if } M \text{ ever prints a non-blank symbol} \\ \quad \text{when started on an all-blank input} \\ \\ \text{undefined otherwise} \end{cases}$$

- This partial function is computable by a TM.

How to Compute nbp:

- The TM N that computes nbp uses the encoding of M to **simulate** M started on an all-blank tape.
- If M were ever to print a non-blank symbol during the simulation, N would stop in an accepting state (or equivalently, write 1 as output).
- If M never prints a non-blank symbol, N only stops when M would have. However, N does not accept (or write 1) in this case.

Universal Turing Machines

- A Turing machine that interprets a description of other TMs in order to carry out the actions specified is called a **Universal Turing Machine** (UTM).

- In effect, the encoded description is a **program** for the UTM.

Existence of Non-Computable Functions

- We can easily see that there are non-computable functions of type $N \rightarrow N$.
- In order for f to be computable, there must be a TM that computes f.
- Every TM can be described as a string encoding.
- The set of all encodings E is countable.
- However, the set of all functions F (even when restricted to range {0, 1}, as in the characteristic functions for subsets of N) is uncountable ($|E| < |F|$).

Specific Non-Computable Functions

- The previous argument shows that non-computable functions exist, but does not construct one.
- The following argument gives the basic non-computable function.
- Other non-computable functions can be shown based upon it.

The Divergence-Testing Partial Function

- Recall that <M> means an encoding of TM M.
- Consider the partial function

$$\text{dtp}(\langle M \rangle) = \begin{cases} 1 \text{ if } M \text{ diverges on input } \langle M \rangle \\ \text{undefined otherwise} \end{cases}$$

- (Note: "divergent" and undefined result are the same thing.)

dtp is Not Computable

- The proof is by contradiction, and uses the LEM.
- Suppose that dtp were computable.
- Let D be a TM that computes dtp.
- Then to **summarize**, for an TM M:
  D with input <M> diverges
  iff M does not diverge on <M>.
- In particular, taking D for M:
  D with input <D> diverges
  iff D does not diverge on <D>.

To belabor the point …

- If D on input <D> diverges, then from the definition of dtp,

$$\text{dtp}(\langle M \rangle) = \begin{cases} 1 \text{ if M diverges on input } \langle M \rangle \\ \text{undefined otherwise} \end{cases}$$

dtp(<D>) = 1.

But then D **can't** diverge, because it has to give result 1.

More belaboring …

- On the other hand, if D on input <D> doesn't diverge, then from the definition of dtp,

$$\text{dtp}(\langle M \rangle) = \begin{cases} 1 \text{ if M diverges on input } \langle M \rangle \\ \text{undefined otherwise} \end{cases}$$

dtp(<D>) is undefined. But D computes dtp, so it **must** therefore **diverge** with input <D>.

Diagonalization Again

- As with the proof of Cantor's theorem, the argument can be seen as a diagonalization.
- Enumerate all the Turing machines as rows and all inputs as columns of an infinite matrix.
- Put a 1 in column if M **converges** on input <M>. Otherwise put div (divergent).
- From the diagonal of this matrix, construct a row by flipping each 1 to div and each div to 1.
- This row can't be a row of the original matrix.
- This row is the characteristic partial function of dtp.

Diagonalization of TM's

	$<M_0>$	$<M_1>$	$<M_2>$	$<M_3>$	$<M_4>$	…
M_0	1	1	1	1	1	…
M_1	1	div	1	1	1	…
M_2	div	div	1	1	1	…
M_3	div	div	1	1	div	…
M_4	div	div	div	1	div	…
…						…
dtp	div	1	div	div	1	