Day15

We are ready to introduce a formal model that will capture a notion of general purpose computation.

- *Similar to DFA, NFA, PDA*: input will be an arbitrary string over a fixed alphabet.
- *Different from NFA, PDA*: machine is deterministic.
- *Different from DFA, NFA, PDA*: read-write head can move both to the left and to the right, and can extend to the right past the original input.
- *Similar to DFA, NFA, PDA*: transition function drives computation one step at a time by moving within a finite set of states, always starting at designated start state.
- *Different from DFA, NFA, PDA*: the special states for rejecting and accepting take effect immediately.

(See more details: Sipser p. 166)

Formally: a Turing machine is $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ where δ is the **transition function**

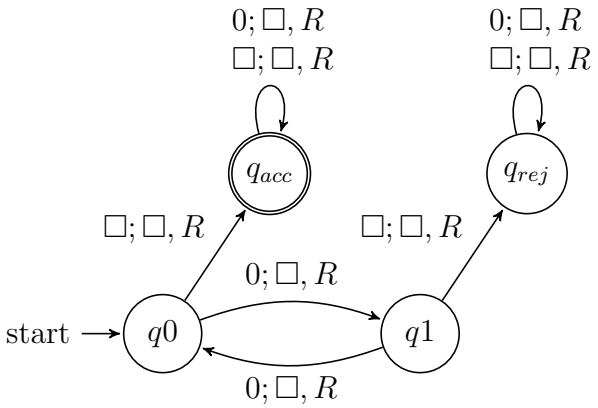
$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$$

The **computation** of M on a string w over Σ is:

- Read/write head starts at leftmost position on tape.
- Input string is written on $|w|$-many leftmost cells of tape, rest of the tape cells have the blank symbol. **Tape alphabet** is Γ with $\textvisiblespace \in \Gamma$ and $\Sigma \subseteq \Gamma$. The blank symbol $\textvisiblespace \notin \Sigma$.
- Given current state of machine and current symbol being read at the tape head, the machine transitions to next state, writes a symbol to the current position of the tape head (overwriting existing symbol), and moves the tape head L or R (if possible).
- Computation ends **if and when** machine enters either the accept or the reject state. This is called **halting**. Note: $q_{accept} \neq q_{reject}$.

The **language recognized by the Turing machine** M, is $L(M) = \{w \in \Sigma^* \mid w \text{ is accepted by } M\}$, which is defined as

$$\{w \in \Sigma^* \mid \text{computation of } M \text{ on } w \text{ halts after entering the accept state}\}$$



Formal definition:

Sample computation:

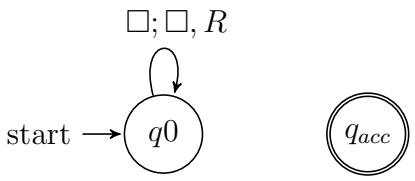
$q0 \downarrow$						
0	0	0	␣	␣	␣	␣

The language recognized by this machine is ...

Describing Turing machines (Sipser p. 185) To define a Turing machine, we could give a

- **Formal definition**: the 7-tuple of parameters including set of states, input alphabet, tape alphabet, transition function, start state, accept state, and reject state; or,
- **Implementation-level definition**: English prose that describes the Turing machine head movements relative to contents of tape, and conditions for accepting / rejecting based on those contents.
- **High-level description**: description of algorithm (precise sequence of instructions), without implementation details of machine. As part of this description, can "call" and run another TM as a subroutine.

Fix $\Sigma = \{0, 1\}$, $\Gamma = \{0, 1, \textvisiblespace\}$ for the Turing machines with the following state diagrams:



Example of string accepted:
Example of string rejected:

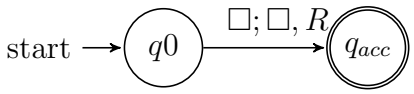
Implementation-level description

High-level description

Example of string accepted:
Example of string rejected:

Implementation-level description

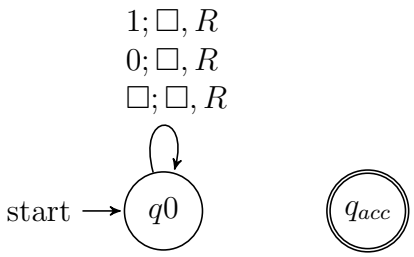
High-level description



Example of string accepted:
Example of string rejected:

Implementation-level description

High-level description



Example of string accepted:
Example of string rejected:

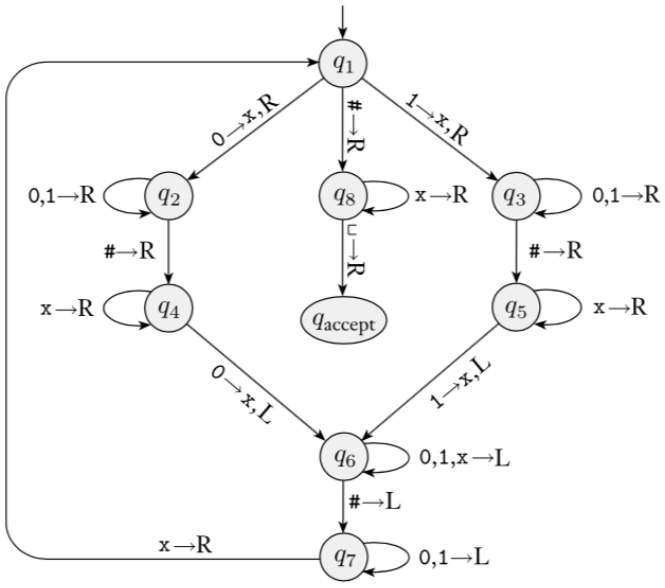
Implementation-level description

High-level description

Day16

Sipser Figure 3.10

Conventions in state diagram of TM: $b \to R$ label means $b \to b, R$ and all arrows missing from diagram represent transitions with output $(q_{reject}, \sqcup, R)$



Implementation level description of this machine:

> Zig-zag across tape to corresponding positions on either side of # to check whether the characters in these positions agree. If they do not, or if there is no #, reject. If they do, cross them off.
>
> Once all symbols to the left of the # are crossed off, check for any un-crossed-off symbols to the right of #; if there are any, reject; if there aren't, accept.

The language recognized by this machine is

$$\{w\#w \mid w \in \{0,1\}^*\}$$

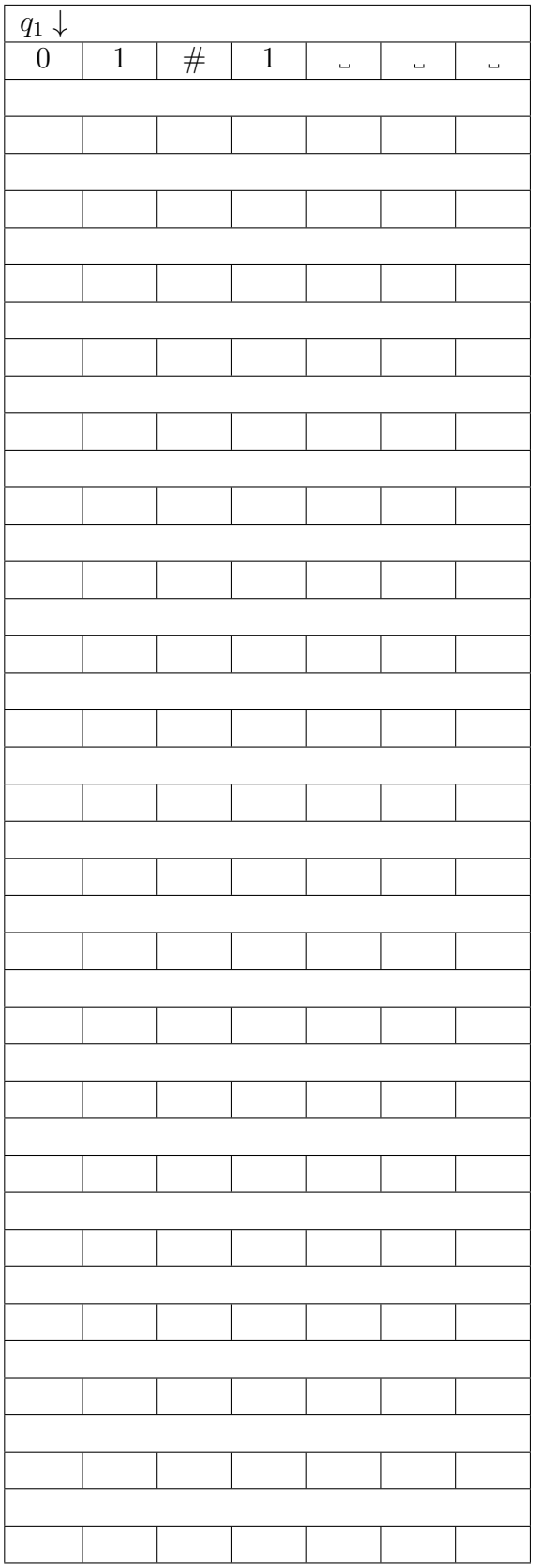
Computation on input string 01#01

$q_1 \downarrow$						
0	1	#	0	1	␣	␣

High-level description of this machine is

Extra practice

Computation on input string 01#1



Recall: High-level descriptions of Turing machine algorithms are written as indented text within quotation marks. Stages of the algorithm are typically numbered consecutively. The first line specifies the input to the machine, which must be a string.

A language L is **recognized by** a Turing machine M means

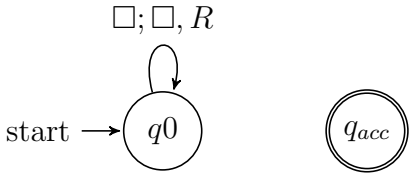
A Turing machine M **recognizes** a language L means

A Turing machine M is a **decider** means

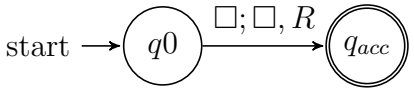
A language L is **decided by** a Turing machine M means

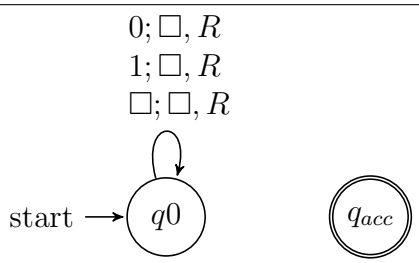
A Turing machine M **decides** a language L means

Fix $\Sigma = \{0, 1\}$, $\Gamma = \{0, 1, \textvisiblespace\}$ for the Turing machines with the following state diagrams:

<table>
<tr>
<td> 

Decider? Yes / No</td>
<td> 

Decider? Yes / No</td>
</tr>
<tr>
<td> 

Decider? Yes / No</td>
<td> 

Decider? Yes / No</td>
</tr>
</table>

Day17

A **Turing-recognizable** language is a set of strings that is the language recognized by some Turing machine. We also say that such languages are recognizable.

A **Turing-decidable** language is a set of strings that is the language recognized by some decider. We also say that such languages are decidable.

An **unrecognizable** language is a language that is not Turing-recognizable.

An **undecidable** language is a language that is not Turing-decidable.

True or **False**: Any decidable language is also recognizable.

True or **False**: Any recognizable language is also decidable.

True or **False**: Any undecidable language is also unrecognizable.

True or **False**: Any unrecognizable language is also undecidable.

True or **False**: The class of Turing-decidable languages is closed under complementation.

Using formal definition:

Using high-level description:

Church-Turing Thesis (Sipser p. 183): The informal notion of algorithm is formalized completely and correctly by the formal definition of a Turing machine. In other words: all reasonably expressive models of computation are equally expressive with the standard Turing machine.

Day18

Definition: A language L over an alphabet Σ is called **co-recognizable** if its complement, defined as $\Sigma^* \setminus L = \{x \in \Sigma^* \mid x \notin L\}$, is Turing-recognizable.

Notation: The complement of a set X is denoted with a superscript c, X^c, or an overline, $\overline{X}$.

Theorem (Sipser Theorem 4.22): A language is Turing-decidable if and only if both it and its complement are Turing-recognizable.

Proof, first direction: Suppose language L is Turing-decidable. WTS that both it and its complement are Turing-recognizable.

Proof, second direction: Suppose language L is Turing-recognizable, and so is its complement. WTS that L is Turing-decidable.

Dovetailing: interleaving progress on multiple computations by limiting the number of steps each computation makes in each round.

Claim: If two languages (over a fixed alphabet Σ) are Turing-decidable, then their union is as well.

Proof:

Claim: If two languages (over a fixed alphabet Σ) are Turing-recognizable, then their union is as well.

Proof:

Day19

The Church-Turing thesis posits that each algorithm can be implemented by some Turing machine.

Describing algorithms (Sipser p. 185) To define a Turing machine, we could give a

- **Formal definition**: the 7-tuple of parameters including set of states, input alphabet, tape alphabet, transition function, start state, accept state, and reject state. This is the low-level programming view that models the logic computation flow in a processor.
- **Implementation-level definition**: English prose that describes the Turing machine head movements relative to contents of tape, and conditions for accepting / rejecting based on those contents. This level describes memory management and implementing data access with data structures.
  - Mention the tape or its contents (e.g. "Scan the tape from left to right until a blank is seen.")
  - Mention the tape head (e.g. "Return the tape head to the left end of the tape.")
- **High-level description** of algorithm executed by Turing machine: description of algorithm (precise sequence of instructions), without implementation details of machine. High-level descriptions of Turing machine algorithms are written as indented text within quotation marks. Stages of the algorithm are typically numbered consecutively. The first line specifies the input to the machine, which must be a string.
  - Use other Turing machines as subroutines (e.g. "Run M on w")
  - Build new machines from existing machines using previously shown results (e.g. "Given NFA A construct an NFA B such that $L(B) = \overline{L(A)}$")
  - Use previously shown conversions and constructions (e.g. "Convert regular expression R to an NFA N")

Formatted inputs to Turing machine algorithms

The input to a Turing machine is always a string. The format of the input to a Turing machine can be checked to interpret this string as representing structured data (like a csv file, the formal definition of a DFA, another Turing machine, etc.)

This string may be the encoding of some object or list of objects.

Notation: $\langle O \rangle$ is the string that encodes the object O. $\langle O_1, \ldots, O_n \rangle$ is the string that encodes the list of objects $O_1, \ldots, O_n$.

Assumption: There are algorithms (Turing machines) that can be called as subroutines to decode the string representations of common objects and interact with these objects as intended (data structures). These algorithms are able to "type-check" and string representations for different data structures are unique.

For example, since there are algorithms to answer each of the following questions, by Church-Turing thesis, there is a Turing machine that accepts exactly those strings for which the answer to the question is "yes"

- Does a string over $\{0, 1\}$ have even length?
- Does a string over $\{0, 1\}$ encode a string of ASCII characters?[1]
- Does a DFA have a specific number of states?
- Do two NFAs have any state names in common?
- Do two CFGs have the same start variable?

A **computational problem** is decidable iff language encoding its positive problem instances is decidable.

The computational problem "Does a specific DFA accept a given string?" is encoded by the language

$$\begin{aligned}&\{\text{representations of DFAs } M \text{ and strings } w \text{ such that } w \in L(M)\}\\ =&\{\langle M, w\rangle \mid M \text{ is a DFA}, w \text{ is a string}, w \in L(M)\}\end{aligned}$$

The computational problem "Is the language generated by a CFG empty?" is encoded by the language

$$\begin{aligned}&\{\text{representations of CFGs } G \text{ such that } L(G) = \emptyset\}\\ =&\{\langle G\rangle \mid G \text{ is a CFG}, L(G) = \emptyset\}\end{aligned}$$

The computational problem "Is the given Turing machine a decider?" is encoded by the language

$$\begin{aligned}&\{\text{representations of TMs } M \text{ such that } M \text{ halts on every input}\}\\ =&\{\langle M\rangle \mid M \text{ is a TM and for each string } w, M \text{ halts on } w\}\end{aligned}$$

Note: writing down the language encoding a computational problem is only the first step in determining if it's recognizable, decidable, or ...

Deciding a computational problem means building / defining a Turing machine that recognizes the language encoding the computational problem, and that is a decider.

Some classes of computational problems will help us understand the differences between the machine models we've been studying. (Sipser Section 4.1)

[1] An introduction to ASCII is available on the w3 tutorial here.

Acceptance problem		
... for DFA	A_{DFA}	$\{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$
... for NFA	A_{NFA}	$\{\langle B, w\rangle \mid B \text{ is a NFA that accepts input string } w\}$
... for regular expressions	A_{REX}	$\{\langle R, w\rangle \mid R \text{ is a regular expression that generates input string } w\}$
... for CFG	A_{CFG}	$\{\langle G, w\rangle \mid G \text{ is a context-free grammar that generates input string } w\}$
... for PDA	A_{PDA}	$\{\langle B, w\rangle \mid B \text{ is a PDA that accepts input string } w\}$
Language emptiness testing		
... for DFA	E_{DFA}	$\{\langle A\rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}$
... for NFA	E_{NFA}	$\{\langle A\rangle \mid A \text{ is a NFA and } L(A) = \emptyset\}$
... for regular expressions	E_{REX}	$\{\langle R\rangle \mid R \text{ is a regular expression and } L(R) = \emptyset\}$
... for CFG	E_{CFG}	$\{\langle G\rangle \mid G \text{ is a context-free grammar and } L(G) = \emptyset\}$
... for PDA	E_{PDA}	$\{\langle A\rangle \mid A \text{ is a PDA and } L(A) = \emptyset\}$
Language equality testing		
... for DFA	EQ_{DFA}	$\{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$
... for NFA	EQ_{NFA}	$\{\langle A, B\rangle \mid A \text{ and } B \text{ are NFAs and } L(A) = L(B)\}$
... for regular expressions	EQ_{REX}	$\{\langle R, R'\rangle \mid R \text{ and } R' \text{ are regular expressions and } L(R) = L(R')\}$
... for CFG	EQ_{CFG}	$\{\langle G, G'\rangle \mid G \text{ and } G' \text{ are CFGs and } L(G) = L(G')\}$
... for PDA	EQ_{PDA}	$\{\langle A, B\rangle \mid A \text{ and } B \text{ are PDAs and } L(A) = L(B)\}$

Example strings in A_{DFA}

Example strings in E_{DFA}

Example strings in EQ_{DFA}

M_1 = "On input $\langle M, w\rangle$, where M is a DFA and w is a string:

0. Type check encoding to check input is correct type. If not, reject.
1. Simulate M on input w (by keeping track of states in M, transition function of M, etc.)
2. If the simulation ends in an accept state of M, accept. If it ends in a non-accept state of M, reject. "

What is $L(M_1)$?

Is M_1 a decider?

Alternate description: Sometimes omit step 0 from listing and do implicit type check.

Synonyms: "Simulate", "run", "call".

True / False: $A_{REX} = A_{NFA} = A_{DFA}$

True / False: $A_{REX} \cap A_{NFA} = \emptyset$, $A_{REX} \cap A_{DFA} = \emptyset$, $A_{DFA} \cap A_{NFA} = \emptyset$

$E_{DFA} = \{\langle A \rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}$. A Turing machine that decides E_{DFA} is

$M_2 =$"On input $\langle M \rangle$ where M is a DFA,

1. For integer $i = 1, 2, \ldots$
2. Let s_i be the ith string over the alphabet of M (ordered in string order).
3. Run M on input s_i.
4. If M accepts, reject. If M rejects, increment i and keep going."

$M_3 =$ " On input $\langle M \rangle$ where M is a DFA,

1. Mark the start state of M.
2. Repeat until no new states get marked:
3. Loop over the states of M.
4. Mark any unmarked state that has an incoming edge from a marked state.
5. If no accept state of M is marked, _____; otherwise, _____".

To build a Turing machine that decides EQ_{DFA}, notice that

$$L_1 = L_2 \qquad \text{iff} \qquad (\ (L_1 \cap \overline{L_2}) \cup (L_2 \cap \overline{L}_1)\ ) = \emptyset$$

There are no elements that are in one set and not the other

$M_{EQDFA} =$

Summary: We can use the decision procedures (Turing machines) of decidable problems as subroutines in other algorithms. For example, we have subroutines for deciding each of A_{DFA}, E_{DFA}, EQ_{DFA}. We can also use algorithms for known constructions as subroutines in other algorithms. For example, we have subroutines for: counting the number of states in a state diagram, counting the number of characters in an alphabet, converting DFA to a DFA recognizing the complement of the original language or a DFA recognizing the Kleene star of the original language, constructing a DFA or NFA from two DFA or NFA so that we have a machine recognizing the language of the union (or intersection, concatenation) of the languages of the original machines; converting regular expressions to equivalent DFA; converting DFA to equivalent regular expressions, etc.