# Halting Problem

Adam Wyner
CS3518, Spring 2017
University of Aberdeen

# Decidability

- Investigate the power of algorithms to solve problems.
- Some problems can be solved, others cannot!
- If you know a problem is algorithmically unsolvable, then it must be simplified before we propose an algorithm:
  - Instead of Hilbert's problem, you might build an algorithm for deciding whether any roots exist between -n and +n.
- Computing has limitations, which we must consider these if we want to use it well.
- Caveat: we focus on decision problems (i.e., yes/no questions). The story for other problems is analogous

# Key ideas

- We know about enumerations and the diagonalisation – sometimes we see we have more problems than solutions.
- If we can *convert* a problem into a form to apply the diagonalisation analysis, then we can see if problems and solutions match up or not.
- Conversion here is going to be 'encode' the problem in a form that diagonalisation can treat.

# Decidable Languages

- Let's look at problems that involve finite automata
  - This will sharpen our intuitions about computability
  - We shall learn some tricks that we shall go on to apply to TMs themselves

- We present mere sketches of proofs

# Regular Languages

- formal languages that can be expressed using regular expressions (string patterns).
- recognised by finite state automata.
  - regular: all strings over {a,b} which contain an even number of as
  - not regular: all strings over {a,b} which contain as followed by the same number of bs. An FSA has a finite memory, so cannot record the number of as.
- Key question – is a given regular language (recognised by the correlated FSA) decidable?

# Recall TM decidability

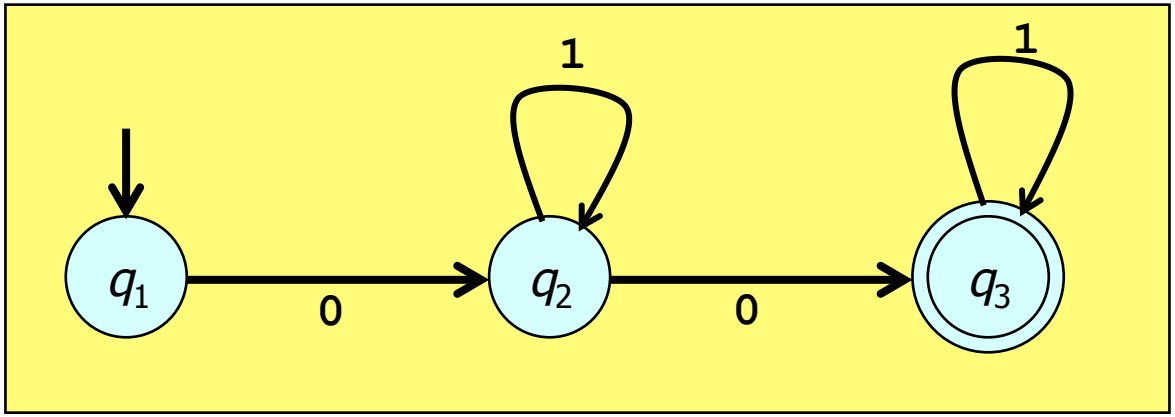
- A language is recognizable iff there is a TM halts and accepts only the strings in that language. For strings not in the language, the TM either rejects them or does not halt at all. That is, it could not stop/decide.
- A language is decidable iff there is a TM which will accept strings in the language and reject strings not in the language. That is, it always stops.

# Recall encodings

- We want to relate FSAs to TMs so as to be able to say of an FSA language that is/is not decidable, which is a TM concept
- In the TM lecture Part 1, there is a discussion of configurations, which is along the lines of viewing a TM as a string; a string representation of computations.

# Decidable Problems on Reg. Langs.

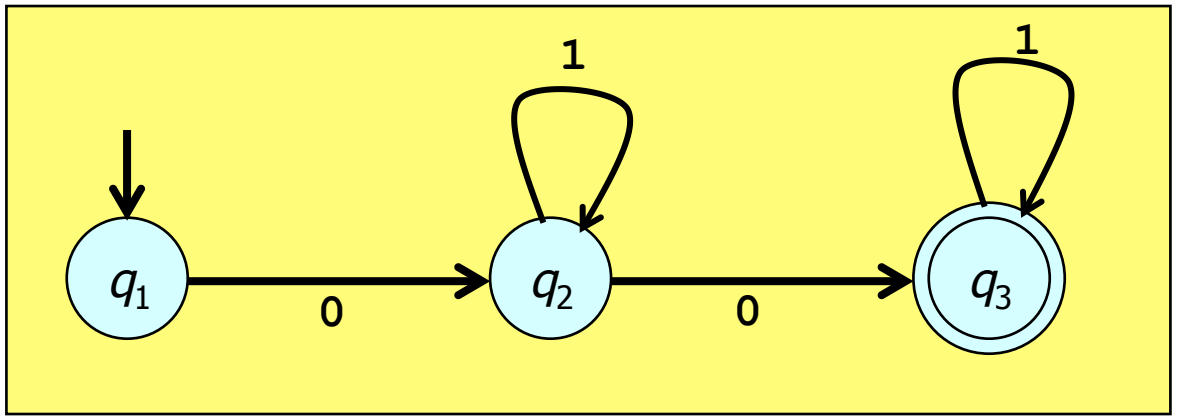
Deterministic Finite Automata (DFAs):




What language does this DFA recognise/accept?

# Decidable Problems on Reg. Langs.

Deterministic Finite Automata (DFAs):




$L = \{01^*01^*\}$

# Decidable Problems on R.L. (Cont'd)

- The acceptance problem for DFAs:

  given

  – A deterministic finite automaton (DFA) and

  – A string

  check if the DFA accepts the string
- Need a "bridge" between DFAs (about strings) and TMs (to determine decidability)
- Create a derivative language which encodes DFAs coupled with strings that the DFAs accept:

  $$A_{DFA} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$$
- "DFA B accepts input w" same as "$\langle B, w\rangle \in A_{DFA}$"

# Decidable Language 1

Theorem: "$A_{DFA}$ is a decidable language"

Proof sketch: The TM M decides $A_{DFA}$

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

1. Simulate B on input w
2. If the simulation ends in an accepting state of B , then accept;
   If the simulation ends in a non-accepting state of B, then reject.

M terminates because the number of steps needed by a DFA to reach accept/reject is finite (bounded by the number of symbols in w)

# Decidable Language 1

The transitions of an DFA can be encoded as follows:

- Each of the states qi is encoded as a string: q1=DA, q2=DAA, q3=DAAA, ...
- Each of the symbols Sj is encoded as a string: S1=DC, S2=DCC, S3=DCCC, ...
- Now each transition rule can be encoded as a string, e.g. d(q1,S2)=q2 is encoded as DADCCDAAE (with E for “end of transition”)

# Simulation of DFA on tape of a TM

- Put the input string w on the tape
- Encode the DFA B on a tape as shown
- Mark the current state
- Mark the position of the head.
- All of this is part of the input string to the TM (separate everything using suitable characters)
- Continued...

# Simulation of DFA on tape of a TM

- Add transition rules to make sure that, e.g.:

  In $q_0$, reading the first symbol w' of w,
  - Find on the tape the (encoded) transition rule that is applicable (i.e., the encoding for a rule of the form $<q_0,w',q_1>$, for some $q_1$)
  - Replace $q_0$ by $q_1$ as the current state
  - Update the position of the head
- If/when no applicable rule is found then reject w
- If/when current state = $q_{acc}$ then accept w

# Decidable Language 2

- Is there an algorithm to check if a DFA accepts any strings at all?
- In terms of languages:

$$E_{DFA} = \{\langle A \rangle \mid A \text{ is a DFA and } L(A) = \varnothing\}$$

# Decidable Language 2 (Cont'd)

Theorem: "$E_{DFA}$ is a decidable language"

Proof: a DFA accepts some string if, and only if, it is possible to go from the start state to an accept state following the arrows

A TM T that decides $E_{DFA}$:

*T* = On input $\langle A \rangle$, where *A* is a DFA:

1. Mark the start state of *A*
2. Repeat until no new states get marked:
3. Mark any state that has a transition coming into it from any marked state
4. If an accept state is marked, accept; otherwise reject

(Why does T terminate?)

# Decidable Language 3

- Is there an algorithm to check if two DFAs are equivalent, i.e., they recognise the same language?
- In terms of languages:

$$EQ_{DFA} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$$

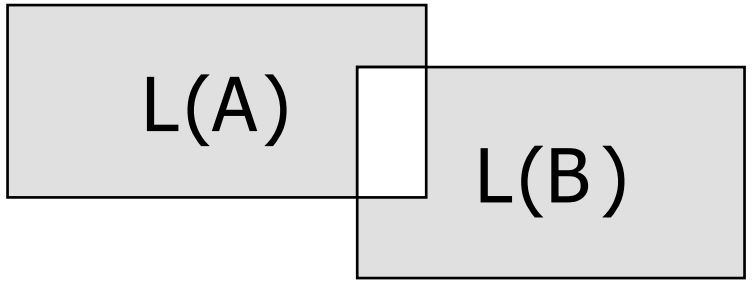
- Let's devise a solution to this problem using the previous TM

# Decidable Language 3 (Cont'd)

- We build a new DFA C from A and B such that
  - C accepts only those strings that are accepted either by A or B but not both;
  - A and B recognise the same language iff C is empty
- The language of C is

$$L(C) = (L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap L(B))$$

  - This is the symmetric difference of L(A) and L(B)
  - $\overline{L(A)}$ is the complement of L(A)
  - Diagrammatically: in grey, L(C)



# Decidable Language 3 (Cont'd)

- We build C from A and B with operations on DFAs:

  Given DFAs for L(X) and L(Y),

  - We can build a DFA that recognises $L(X) \cap L(Y)$
  - We can build a DFA that recognises $\overline{L(X)} \cup L(Y)$
  - We can build a DFA that recognises complement of L(X)

  (Check that these three claims are true!)

  We can therefore build a DFA that recognises C
- Once we have built C, we can use previous TM to check if L(C) is empty
  - if L(C) is empty then L(A) = L(B)

# Decidable Language 3 (Cont'd)

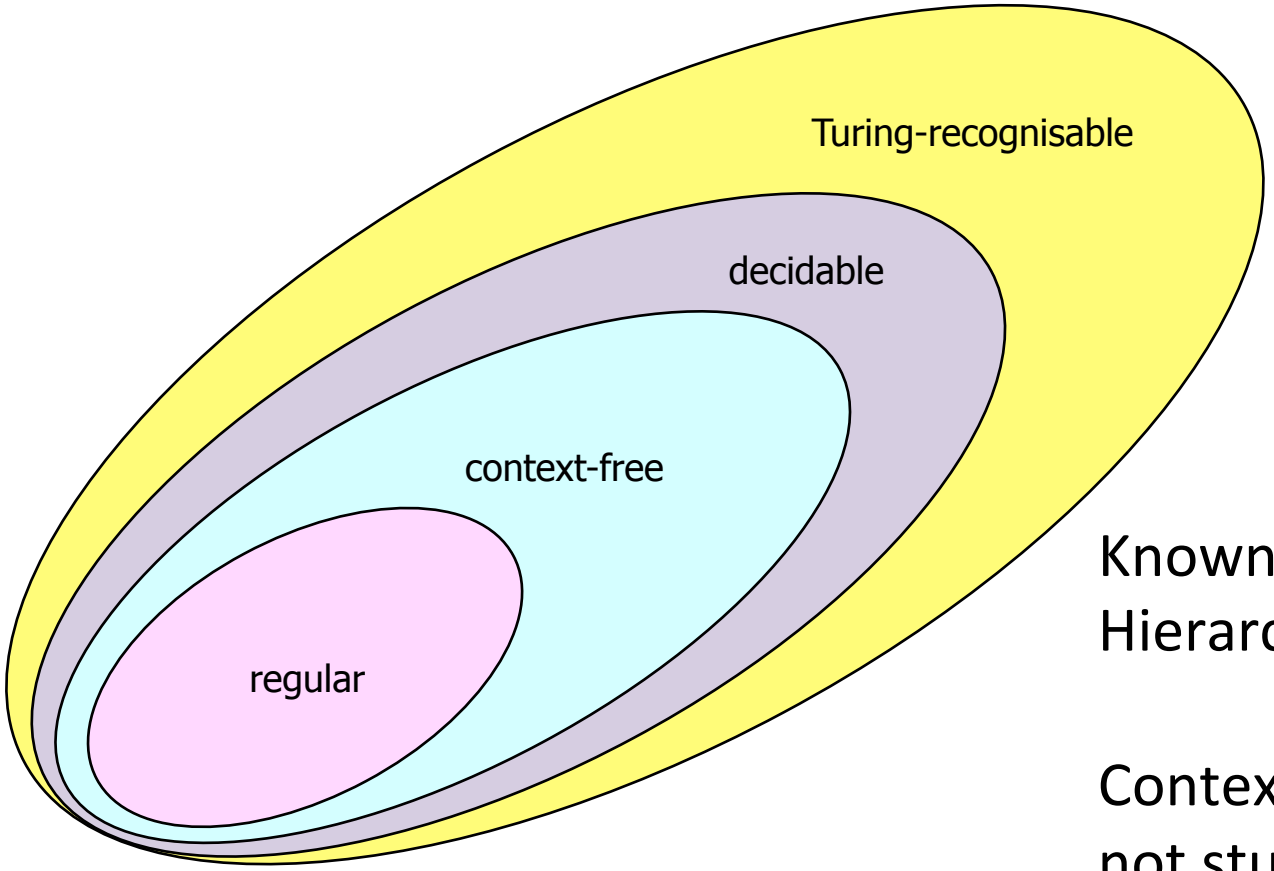
The proof is a TM:

F = On input $\langle A,B \rangle$ where A and B are DFAs:

1. Build DFA C as described
2. Run TM T from previous example on $\langle C \rangle$
4. If T accepts, accept; If T rejects, reject

# A Hierarchy of Languages

Classes of languages and how they relate:




Known as the Chomsky Hierarchy

Context-free languages: not studied in this course

# Undecidable problems

- The theory of computability is interesting because some problems/languages are not computable
- An undecidable problem is a decision problem where there isn't an algorithm which gives a correct yes or no answer
- We shall first prove in the abstract that some problems are not computable
- Then we focus on one specific problem, the halting problem, showing that it is not computable

- Note that we encode strings as natural numbers, which means that we can reuse earlier ideas about enumerations.

# Theorem: there exist uncomputable problems

Outline of proof:

1. There are uncountably many languages/problems
2. There are countably many TMs, because each TM can only recognise a single language
3. There are more languages than TMs
4. Conclusion:
   - Some languages are not recognised by any TM.
   - That is: some languages are not Turing-recognisable

Let's look at this argument in more detail

# More Languages than TMs

First: There are countably many TMs:

1. Each of the countably many states qi is encoded as a string: q1=DA, q2=DAA, q3=DAAA, ...
2. Each of the symbols Sj is encoded as a string: S1=DC, S2=DCC, S3=DCCC, ...
3. Now each transition rule can be encoded as a string. E.g. d(q1,S2)=q2,S4,R is encoded as DADCCDAADCCCCRE (with E for "end of transition")
4. Replace each letter by a digit: the result is a positive integer.
5. The TM is represented by stringing together these integers. Each such string of integers (an integer) represents at most one TM.

# More Languages than TMs

- The same argument shows that there are only countably many possible
  - JAVA programs
  - web pages
  - novels
  - …
- Essentially, this is because each of these has a finite alphabet and a finite (though unlimited!) length

# More Languages than TMs

Second: There are uncountably many problems/languages

- This is most easily shown by focussing on a particular class of problems. For example,
  - Consider all functions f :: Int -> Int
    Each defines a language of 3-tuples.
    For example $(f_1,x,y)$ iff $f_1(x)=y$
  - The number of functions of this type is uncountable.
    Prove this using Cantor's diagonal argument: Try to enumerate all functions $f_i$; each $f_i(j)$ is defined for every integer j. Define a new function g as $g(j)=f_i(j + 1)$.
    g cannot be an element of the enumeration since it is different from any $f_i$.

# Uncountably many f :: Int -> Int

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | … |
|---|---|---|---|---|---|---|---|---|---|
| f1 | . | | | | | | | | |
| f2 | | . | | | | | | | |
| f3 | | | . | | | | | | |
| f4 | | | | . | | | | | |
| ... | | | | | | | | | |

g differs from f1 on the argument 1

g differs from f2 on the argument 2

... *etc.* **So: g is not in the enumeration!**

# Where we are now

- You've seen a very abstract proof that some languages are not Turing-decidable.
  - In laymen's terms: some problems are not computable
- More specifically, you've seen that some functions
  f :: Int → Int  cannot be programmed using a TM
- But we haven't shown you a concrete example (e.g., a concrete problem for which there is no algorithm).

- We now focus on one example: the halting problem