# Module 9

## Undecidability

### What computers cannot do.

CS 360: Introduction to the Theory of Computing

Collin Roberts, University of Waterloo

9.1

### Topics for this module

- An undecidable language
- Other undecidable languages
- Reduction: how to prove a given language is undecidable
- Undecidable problems in automata theory
- End of the course

In my view this is by far most important and interesting material in CS 360. 9.2

## 1 An undecidable language

### A language that is not decidable

Remember from before:

- $L$ is decidable or recursive if there exists a Turing machine $M$ with $L = L(M)$ where $M$ halts on all inputs.
- $L$ is recursively enumerable if there exists a Turing machine $M$ with $L = L(M)$.

Are all languages recursive?
No. 9.3

### Diagonalization argument

The argument is similar to showing there are more real numbers than integers.
This is counterintuitive: is one type of infinity bigger than another?
Yes.

- We say that sets $S$ and $T$ are of equal cardinality if there exists a bijection $f$ between $S$ and $T$.
- (Recall that a function $f : S \to T$ is a bijection if it is both injective and surjective.)
- If $S$ and $T$ are both finite, then the definition of equal cardinality agrees with our intuition: the sets are of equal cardinality if they have the same number of elements.
- For infinite sets, this can feel counter-intuitive: for example, if
  - $S = \{\text{even integers}\}$, and
  - $T = \{\text{all integers}\}$, and
  - $f(x) = \frac{x}{2}$,

  then $f : S \to T$ is a bijection.
- $S$ and $T$ are of equal cardinality, even though our intuition might tell us that $S$ is "half" the size of $T$.

Are there bigger and smaller infinities? 9.4

## Countable sets

- A set $S$ is countably infinite: of equal cardinality to $\mathbb{Z}$ (or any infinite subset of $\mathbb{Z}$).
- A set $S$ is countable: of equal cardinality to a subset of $\mathbb{Z}$.
- A countable set $S$ can be described by listing: $S = \{s_1, s_2, s_3, \ldots, s_n\}$ for finite $S$, or $S = \{s_1, s_2, s_3, \ldots\}$ for infinite $S$.
- A set $S$ is uncountable if it is not countable.
- If $S$ is uncountable, then any listing $S = \{s_1, s_2, s_3, \ldots\}$ misses some members of $S$.

We know that countable sets exist, e.g. $\mathbb{Z}$, or { even integers }. The positive integers are also countable.

But do uncountable sets exist? 9.5

## The real numbers between 0 and 1 are uncountable.

Consider $S = [0,1] \subset \mathbb{R}$. Is $S$ countable?

- For a contradiction, assume that $S$ is countable.
- Suppose that $f : \mathbb{Z}^+ \to S$ is a bijection.
- Then $S = \{s_1, s_2, \ldots\} = \{f(1), f(2), \ldots, \}$.
- List the binary expansions of $S$, in the order $s_1, s_2, \ldots$.
- Consider the number $a$ that we get by concatenating the opposite of the $i$th bit at each position $s_i$ in the list.

$$\begin{array}{rcl}
s_1 & = & .\mathbf{0}11011010111\cdots \\
s_2 & = & .0\mathbf{1}0001111111\cdots \\
s_3 & = & .11\mathbf{1}000011101\cdots \\
s_4 & = & .110\mathbf{1}11100101\cdots \\
 & \vdots &
\end{array}$$

In this example, $a = .1000\cdots$. Note: $a \in S$, by the definition of $S$. 9.6

## The real number $a$ is not one of the $s_i$!

We have constructed $a$ so that it is not one of the $s_i$.

- It differs from $s_1$ in the first bit.
- It differs from $s_2$ in the second bit.
- It differs from $s_i$ in the $i$th bit!

We see $a \neq s_i$ for all choices of $i$. But $a \in S$. So we did not enumerate all of the members of $S$! We have a contradiction, and therefore $S$ is not countable.

- Note: if we add $a$ to the list, we can again find a missing number.
- To show this takes some boring housekeeping, e.g. $0.01111111\cdots = 0.1000000\cdots$.
- Actually, $\mathbb{R}$ is much larger than $\mathbb{Z}$. Take a measure theory course to learn all the gory details.
- Do you find this proof disturbing? If so, then you are not alone! Georg Cantor, who first gave this proof, and created modern set theory, suffered depression and poor psychological health as a result.

This is called a diagonalization argument. 9.7

## Building to a proof about Turing machines

We will adapt this argument to show that there are undecidable languages.

We need to identify each Turing machine $M$ (over the binary alphabet $\{0,1\}$) with a binary string.

- Suppose we have states $q_1, \ldots, q_r$, for some $r$.
- Let $q_1$ be the start state and let $q_2$ be the unique accept state. (One accept state is enough as we halt when we enter any accept state.)

- Suppose that the tape symbols are $X_1, \ldots, X_s$, for some $s$. Assume that
  - $X_1 = 0$,
  - $X_2 = 1$,
  - $X_3 = B$,
  - and the other symbols may be assigned as required.
- Refer to direction $L$ as $D_1$, and $R$ as $D_2$.
- Now we can encode the transition function. 9.8

## Building to a proof about Turing machines

- Suppose that one transition rule is $\delta(q_i, X_j) = (q_k, X_\ell, D_m)$, for some integers $i, j, k, \ell$ and $m$.
- Encode this rule as $0^i 1 0^j 1 0^k 1 0^\ell 1 0^m$.
- As all of $i, j, k, \ell$ and $m$ are at least 1, there are no consecutive 1s in this string.
- A code for the entire TM $M$ consists of all the $n$ codes for the transitions, in some order, separated by pairs of 1s:

$$11C_1 11C_2 11 \cdots 11C_n 11,$$

  where each $C_i$ is the code for transition number $i$ in $M$.
- We have encoded a binary string to identify the Turing machine $M$; treat it as an integer index for each machine $M$. $f(M)$ is the integer that we produce in this way.
- There can be multiple codes for $M$ (but only finitely many). We could renumber the states, for example. We will see soon why this point does not affect the diagonalization argument that we are going to make.
- The set of Turing machines is countable. We just built a bijection to a subset of $\mathbb{Z}$. 9.9

## So what about Turing machines?

- There are only countably many Turing machines, hence there are only countably many recursively enumerable languages.
- But there are uncountably many unary languages.
- Here is a bijection from the set of languages over the alphabet $\{a\}$ to the set $S = [0,1]$ defined before, which we proved is not countable:
- Given a language $L \subseteq \{a\}^*$, define

$$f(L) = \sum_{i \geq 0 : a^i \in L} 2^{-(i+1)}.$$

- $f$ is injective.
  - If $a^i \in L_1$ and $a^i \notin L_2$, then the $(i+1)^{th}$ bit is 1 in $f(L_1)$ but is 0 in $f(L_2)$.
- We can also see how to construct the language $L$ so that $f(L)$ equals any desired expansion, so that $f$ is surjective.
  - If the $i$th bit is 1 in the expansion, then include $a^{i-1}$ in $L$.
  - If the $i$th bit is 0 in the expansion, then exclude $a^{i-1}$ from $L$.
- There must be non-recursively enumerable languages, as there are not enough Turing machines!

This proof is completely unsatisfying! What are these non-recursively enumerable languages? Can we adapt Cantor's proof somehow? 9.10

## A language about Turing machines

Every Turing machine $M$ has an identifier $w = f(M)$.

- Consider the language

$$L_{SA} = \{w \mid \text{ the Turing machine } M \text{ represented by } w \text{ accepts } w\}.$$

- $L_{SA}$ is the language of identifiers for machines that accept when given their own identifiers as inputs.
- $L_{SA}$: The subscript $SA$ indicates self-accepting.
- It might seem that we should not be allowed to make this definition, but why is this? Lots of programs process other programs! Remember: $w = f(M)$ is basically the code of the program for $M$.
  - Parsers, compilers, interpreters, profilers, etc.
  - For example, a syntax parser whose input is the parser should accept: "Yes, that program uses proper syntax."

It will turn out that $L_{SA}$ is not decidable, but $L_{SA}$ will not be our first example of an undecidable language. 9.11

## An undecidable language

Instead, consider

$$L_{NSA} = \{w \mid \text{ the Turing machine } M \text{ represented by } w \text{ does not accept } w\}.$$

- $L_{NSA}$ is the language of identifiers for Turing machines that do not accept when given their own identifiers as inputs.
- For example, a program that finds syntax errors: it will not accept itself, since it does not have any syntax errors! If $M$ is a properly coded syntax error detector represented by $w$, then $w \in L_{NSA}$.

Is $L_{NSA}$ the language of any Turing machine?

- Suppose that $L_{NSA} = L(M)$, for some Turing machine $M$.
- Does $M$ accept its own identifier $w$? Is $w \in L_{NSA}$?
  - Suppose $M$ accepts its own identifier $w$.
    * Then $w$ is not in $L_{NSA}$, which is the language of $M$.
    * So $M$ does not accept $w$.
  - Suppose $M$ does not accept its own identifier $w$.
    * Then $w$ is in $L_{NSA}$, which is the language of $M$.
    * So $M$ accepts $w$.

9.12

## $L_{NSA}$ is not the language of any Turing machine

This tells us that $L_{NSA}$ is not the language of any Turing machine.

- We formalize this via a retread of Cantor's diagonalization argument.
- For any Turing machine $M$, $L_{NSA}$ differs from $L(M)$ in at least one position.
  - If the Turing machine $M$ represented by $w$ accepts $w$ (so that $w \in L(M)$), then $w \notin L_{NSA}$.
  - If the Turing machine $M$ represented by $w$ does not accept $w$ (so that $w \notin L(M)$), then $w \in L_{NSA}$.
- $L_{NSA}$ differs from the languages of all Turing machines.
- By definition, $L_{NSA}$ is not recursively enumerable.
- Then $L_{NSA}$ is also not decidable.

(Your text denotes $L_{NSA}$ as $L_d$, where the subscript $d$ reminds us of diagonalization.)

- Now we see why the uniqueness of the identifier for a given Turing machine is not crucial for this result.
- No identifier can represent a Turing machine $M$ such that $L(M) = L_{NSA}$. 9.13

# 2 Other undecidable languages

## Other undecidable languages

- Some undecidable languages are recursively enumerable.
- Consider $L_{SA}$, the language of encodings for machines that do accept their encoding.
- $L_{SA}$ is recursively enumerable, but undecidable.
- We need a new Turing machine, the Universal Turing Machine: an interpreter.

9.14

## Why have an interpreter?

Why an interpreter?

- Turing machines are a good representation of programs: we build compilers and interpreters in normal computer languages.
- Turing machines can comfortably manipulate other Turing machines' descriptions.

Also historically significant:

- The idea of an interpreter is striking, since people were not actually programming computers yet when Turing had the idea.
- When there were real computers, 10 years later, Turing invented the first real programming language.
- (Also a good machine coder, but talked about using programming languages in debugging.)

9.15

## What will the Universal Turing Machine do?

The Universal Turing Machine $U$ simulates a Turing machine $M$.

- Input: a pair of strings, $(e,w)$. (If you like, the machine's tape alphabet can have parentheses and commas.)
- If $e$ is not the encoding of any Turing machine, then $U$ rejects $(e,w)$.
  - Many identifiers correspond to no Turing machine at all, e.g. 00110 does not start with 1, and
  - 11001011101001010011 is not valid because it has three consecutive 1s.
- If $e = f(M)$ for some Turing machine $M$, and $M$ accepts $w$, then $U$ accepts $(e,w)$.
- If $e = f(M)$ and $M$ rejects $w$, then $U$ rejects $(e,w)$.
- If $e = f(M)$ and $M$ runs forever on $w$, then $U$ runs forever on $(e,w)$.

Define $L_u = L(U)$: the universal language; it includes all pairs $(e,w)$, where

- $e = f(M)$ for some Turing machine $M$, and
- $w \in L(M)$.

9.16

## Does $U$ exist?

$U$ will have four tapes:

- One tape keeps $(e,w)$, which really is $(M,w)$
- One tape maintains the tape for $M$
- One tape maintains the current state $q$ of $M$
  - (state $q_i$ is indicated by $0^i$)
- One tape is used for scratch work

To make a transition in $M$, we:

- Rewind $U$'s first tape, and find the right transition in $M$ for $(q,a)$, where the second tape head points to $a$.
- Important: we can find the correct values for $\delta(q,a)$ on the first tape (that binary string encodes all the logic from $M$ after all).
- Copy the new state for the machine onto the third tape.
- Update the second tape and move the tape head.

9.17

### Some last details

- If we reach an accept state in $M$, then $U$ accepts immediately.
- If $M$ crashes, then $U$ must crash as well.
- We will need a Turing machine to parse a possible identifier for a Turing machine to determine whether it is proper.
- As we know how the identifiers are constructed, we can write an algorithm for this.
- By the Church-Turing Thesis, we can create a Turing machine to execute the algorithm.

9.18

### What good is this?

We may simulate one step of $M$'s execution by many steps of $U$.

- We only care that this simulation can be done, not how slowly it will run.

An interesting idea: we can run $U$ on $U$'s encoding. $U$ is a Turing machine, after all. 9.19

### Is $L_u$ decidable?

The universal language, $L_u$ is recursively enumerable.

- Trivial: $U$ is a Turing machine; its language is r.e.

$L_u$ is not decidable.

- For a contradiction, suppose that $L_u$ is decidable. Suppose there is a Turing machine $U'$ which decides $L_u$.
- Recall that $L_{NSA}$: machine descriptions $w$ of machines $M$ that do not accept $w$ as input, is not r.e., and thus not decidable.
- We will use $U'$ to construct a new Turing machine $M_{NSA}$ that decides membership of $w$ in $L_{NSA}$:
  - Run $U'$ on input $(w,w)$.
  - If $U'$ accepts $(w,w)$, then reject.
  - If $U'$ rejects $(w,w)$, then
    * If $w$ is the encoding of a Turing machine (which we can test), then accept.
    * Else reject.
  - By our assumption, there is no possibility that $U'$ runs forever.

9.20

### What does that get us?

We still need to argue that $M_{NSA}$ decides membership in $L_{NSA}$.

- Suppose that $w$ represents some Turing machine $M$.
- If $M$ accepts $w$, then $U'$ accepts $(w,w)$. So $M_{NSA}$ rejects $w$, as it should.
- If $M$ does not accept $w$, then $U'$ rejects $(w,w)$. So $M_{NSA}$ accepts $w$, as it should.

Therefore $M_{NSA}$ decides membership in $L_{NSA}$. But $L_{NSA}$ is not decidable! This is not possible! Therefore $U'$ cannot exist. This contradiction proves that $U$ is not decidable.
We have proved that the universal language, $L_u$, is recursively enumerable, but not decidable.

9.21

### Reduction

This argument shows a key idea which is used two places in computer science:

1. Producing an algorithm: to solve problem $B$, if you have a solution for $A$ and a way to transform $B$ into $A$, then you are done.
2. Proving no algorithm exists: to show $B$ cannot be solved, if you have shown $A$ cannot be solved, and that $A$ can be transformed into $B$, then $B$ cannot be solved.
   Why? If $B$ had a solution, then we could solve $A$ by transforming $A$ to $B$, and solving $B$.

Use #1 of reduction is common in algorithm design. In computability and complexity, we use #2. 9.22

## This reasoning pops up elsewhere

Fact: perpetual motion machines do not exist.

- If someone says "My bike wheel will never stop spinning"
- You say "If so, then your bike wheel is a perpetual motion machine. But perpetual motion machines do not exist. Therefore, you are lying. Your bike wheel will eventually stop."
- Notice how this works:
  - Someone says "$A$ exists"
  - If $A$ existed, then $B$ would also have to exist.
  - But $B$ does not exist.
  - Hence, $A$ does not exist, either.

9.23

## Another recursively enumerable language

$L_{SA}$, which contains descriptions for Turing machines that accept their own descriptions is recursively enumerable, but not decidable.

- $L_{SA}$ is r.e.
  - Construct a Turing machine $M$, which, on input $w$,
  - runs $U$ on $(w,w)$,
  - accepts if $U$ accepts $(w,w)$, and
  - rejects if $U$ rejects $(w,w)$.
  - Note, $U$ can run forever on input $(w,w)$, causing $M$ to run forever on input $w$.
  - Then by the definition of $U$, it is clear that $L(M) = L_{SA}$.
- $L_{SA}$ is not decidable.
  - If it were, then we could decide membership in $L_{NSA}$ easily:
  - Check membership in $L_{SA}$ and do the opposite.
  - but since $L_{NSA}$ is not decidable, this is impossible.

This give us some closure rules.

9.24

## Closure rules for decidable and recursively enumerable languages

Theorem: If $L$ is decidable, then so is its complement $L'$.

- Proof: Suppose that a Turing machine $M$ decides membership in $L$.
- Create a new Turing machine $M'$ to decide membership in $L'$:
  - For any candidate word $w$,
  - Run the machine $M$ on input $w$,
  - If $M$ accepts $w$, then $M'$ rejects it;
  - if $M$ rejects $w$, then $M'$ accepts it.
  - By our hypothesis, there is no possibility that $M$ runs forever.
- From this construction it is clear that $M'$ decides membership in $L'$.

Remark: This result is false if we replace 'decidable' with 'recursively enumerable', as we shall see on the next slide.

9.25

### Closure rules for decidable and recursively enumerable languages

Theorem: If both $L$ and $L'$ are recursively enumerable, then $L$ is recursive.

- Proof: Suppose that $M$ accepts $L$ and $M'$ accepts $L'$.
- Create a 3-tape Turing machine, $M_L$, to simulate running $M$ and $M'$ in parallel, in which:
  - tape #1 controls whether the machine is currently simulating a step of $M$ or a step of $M'$ (after executing a step in one machine, it alternates to the other),
  - tape #2 simulates the tape of $M$, and
  - tape #3 simulates the tape of $M'$.
- For any word $w$, either $M$ or $M'$ (and not both) must accept $w$ in a finite number of steps.
  - If $M$ accepts $w$, then $M_L$ accepts $w$.
  - If $M'$ accepts $w$, then $M_L$ rejects $w$.
- Then $L$ is decided by $M_L$.

Remark: If the complement of every recursively enumerable language was recursively enumerable, then this Theorem would imply that every recursively enumerable language is decidable, and we already have examples that show that this is not true. 9.26

### A couple of other rules

Theorem: The intersection of two r.e. languages is r.e..

- Proof: Suppose that $M_1$ accepts $L_1$ and $M_2$ accepts $L_2$.
- We will construct a Turing machine $M$ which accepts $L_1 \cap L_2$.
- For any candidate word $w$,
- Run $M_1$ with input $w$.
- If $w \in L_1$, then $M_1$ will accept $w$ in finite time.
- If $M_1$ accepts $w$, then run $M_2$ with input $w$.
- If $M_2$ also accepts $w$, then $M$ accepts $w$.
- If either $M_1$ or $M_2$ rejects $w$, then $M$ rejects $w$.
- If either $M_1$ or $M_2$ runs forever on input $w$, then $M$ runs forever on input $w$.
- It is clear from construction that $M$ accepts $L_1 \cap L_2$.

9.27

### A couple of other rules

Theorem: The union of two r.e. languages is r.e.:

- Proof: Suppose $L(M_1) = L_1$ and $L(M_2) = L_2$.
- Suppose that $e_1$ identifies $M_1$ and $e_2$ identifies $M_2$.
- We will construct a Turing machine $M$ which accepts $L_1 \cup L_2$.
- $M$ is non-deterministic.
- For any candidate word $w$,
- $M$ runs two copies of the universal machine $U$ nondeterministically in parallel.
  - If $U$ accepts either $(e_1, w)$ or $(e_2, w)$, then $M$ accepts $w$.
- It is clear from construction that $M$ accepts $L_1 \cup L_2$.

9.28

## 3 Reductions

### Proper definition of reduction

- Suppose we have two decision problems $P_1$ and $P_2$.
  - Both problems are actually questions of testing membership in a language.
  - In detail, for two languages $L_1$ and $L_2$ and two words $w$ and $x$,

    * $P_1$ is of the form "is $w \in L_1$?", and
    * $P_2$ is of the form "is $x \in L_2$?".
- Suppose also that we have an algorithm $A$ that transforms instances of $P_1$ into instances of $P_2$ such that:
  - "Yes" instances of $P_1$ get mapped to "yes" instances of $P_2$.
  - "No" instances of $P_1$ get mapped to "no" instances of $P_2$.
  - The algorithm $A$ always takes finite time.
- Then we say that $A$ reduces $P_1$ to $P_2$.

We will use reduction to show that problems are undecidable or not recursively enumerable.

9.29

### Using a reduction to prove a language is undecidable

Theorem 9.7: If there is a reduction from $P_1$ to $P_2$, then

1. If $P_1$ is undecidable, then $P_2$ is also undecidable.
2. If $P_1$ is non-recursively enumerable, then $P_2$ is also non-recursively enumerable.

Proof: First part

- For a contradiction, suppose that $P_2$ is decidable.
- Construct a Turing machine to decide $P_1$: for any instance $w$ of $P_1$,
- Apply the reduction algorithm to turn $w$ into an instance $x$ of $P_2$.
- Run the machine that decides "is $x$ in $P_2$?"
- If the answer is "yes", then the answer for "is $w$ in $P_1$?" is "yes".
- If the answer is "no", then the answer for "is $w$ in $P_1$?" is "no".
- So whatever answer we obtained for "is $x$ in $P_2$?" is also the correct answer to "is $w$ in $P_1$?".
- But then $P_1$ is decidable, and this is a contradiction.

9.30

### Using a reduction to prove a language is not recursively enumerable

Second part

- For a contradiction, suppose that $P_2$ is recursively enumerable.
- Construct a Turing machine to accept $P_1$: for any instance $w$ of $P_1$,
- Apply the reduction algorithm to turn $w$ into an instance $x$ of $P_2$.
- Run the machine that tests "is $x$ in $P_2$?"
- If the answer is "yes", then the answer for "is $w$ in $P_1$?" is "yes".
- If the answer is "no", then the answer for "is $w$ in $P_1$?" is "no".
- The machine may also run forever.
- So whatever answer we obtained for "is $x$ in $P_2$?" is also the correct answer to "is $w$ in $P_1$?".
- But then $P_1$ is recursively enumerable, and this is a contradiction.

9.31

### Other undecidable problems about Turing machines

Rice's Theorem:

- Given: any "interesting" property $P$, held by some, but not all recursively enumerable languages.
- Let $L_P$ be the language of Turing machine codes $w = f(M)$ for machines whose language satisfy property $P$.
- Then $L_P$ is not decidable.

We must define "interesting" precisely. A few examples before we do that:

- Empty language: Given a Turing machine $M$, does it accept $\emptyset$?
- Finite language: Given a Turing machine $M$, is $L(M)$ finite?
- Regular language: Given a Turing machine $M$, is its language regular?

9.32

### Proof for non-empty language

Nonempty language: Given $w$ identifying the Turing machine $M$, is the language of $M$ non-empty?

More formally: Let

$$L_{ne} = \{w \mid w \text{ identifies the Turing machine } M \text{ and } L(M) \neq \emptyset\}.$$

$L_{ne}$ is recursively enumerable. Here is an algorithm for a Turing machine $M_{ne}$ that accepts $L_{ne}$:

- Given the identifier $w$ for the Turing machine $M$, guess a word $x$ (nondeterministically) that $M$ might accept.
- (As our alphabet is finite, we can systematically test all possible words starting with the shortest ones first.)
- Nondeterministically execute $M$ on all possible choices of $x$.
- If $M$ accepts any choice $x$, then $M_{ne}$ accepts $w$.

9.33

### Is $L_{ne}$ decidable?

$L_{ne}$ is not decidable. We give two proofs here, the first with "bare hands", the second using Theorem 9.7. The content is essentially the same in both proofs. Not surprisingly, the proof using Theorem 9.7 is a bit shorter.
For both proofs, let $\Sigma$ be a non-empty finite alphabet (e.g. $\Sigma = \{0,1\}$).
"Bare Hands" Proof:

- For a contradiction, suppose that we have a machine, $M_{ne}$, which decides membership in $L_{ne}$.
- From this, we will create a new machine, $M_u$, which decides membership in $L_u$.
- Suppose we have a pair $(e,w)$ whose membership in $L_u$ we want to test.
- Assume that $e$ is the identifier for some Turing machine $M$.
- We lose no generality here, as we know how to decide whether this is true.

9.34

### Is $L_{ne}$ decidable?

- Create a new Turing machine $M'$:
  - $M'$, when called with any input $x$, runs $M$ with input $w$.
  - If $M$ accepts $w$, then $M'$ accepts $x$.
  - If $M$ rejects $w$, then $M'$ rejects $x$.
  - $M$ might also run forever on input $w$.
  - Thus we have that
    $$L(M') = \begin{cases} \Sigma^* & \text{if } M \text{ accepts } w \\ \emptyset & \text{otherwise} \end{cases}$$
  - Note: $M'$ ignores its input $x$. $M'$ always runs $M$ on input $w$.
  - Let $w'$ represent $M'$.
- Now construct $M_u$ to run $M_{ne}$ with input $w'$.
  - If $M_{ne}$ accepts $w'$, then by construction $M$ accepts $w$, and therefore $(e,w) \in L_u$.
  - If $M_{ne}$ rejects $w'$, then by construction $M$ does not accept $w$, and therefore $(e,w) \notin L_u$.
  - By our assumption, $M_{ne}$ halts on every input.
- This shows that $M_u$ decides membership in $L_u$.
- As $L_u$ is undecidable, we have our contradiction.

9.35

## Is $L_{ne}$ decidable?

Proof Using Theorem 9.7:

- Define $P_1$: Is $(e,w)$ in $L_u$?, and $P_2$: Is $w'$ in $L_{ne}$?
- The following algorithm reduces $P_1$ to $P_2$:
  - Let $(e,w)$ be an arbitrary instance for $P_1$.
  - Construct a new Turing machine, $M'$, such that, for any input $x \in \Sigma^*$, $M'$ will run $U$ on $(e,w)$.
  - If $U$ accepts $(e,w)$, then $M'$ accepts $x$.
  - If $U$ rejects $(e,w)$, then $M'$ rejects $x$.
  - $U$ may also run forever on $(e,w)$.
  - Then we have
$$L(M') = \begin{cases} \Sigma^* & \text{if } U \text{ accepts } (e,w) \\ \emptyset & \text{otherwise} \end{cases}$$
  - Let $w'$ represent $M'$.
  - Now take $w'$ as the corresponding instance for $P_2$.
- Then "yes" instances of $P_1$ are sent to "yes" instances of $P_2$, and "no" instances of $P_1$ are sent to "no" instances of $P_2$.
- We have reduced membership testing in $L_u$, the universal language, to membership testing in $L_{ne}$.
- As $L_u$ is undecidable, therefore by Theorem 9.7, so is $L_{ne}$.

9.36

## A few other terminology notes

Church-Turing thesis:

- Any reasonable model of digital computation can be expressed in terms of Turing machines.

Remarks:

- We may think of an algorithm as a Turing machine that computes a function or decides a language.
- There are always a finite number of steps in its computation.
- Recall that undecidable languages, or problems corresponding to membership in undecidable languages, do not have algorithms.

9.37

## What we did for $L_{ne}$

We reduced $L_u$ to $L_{ne}$.

- "Yes" instances of $L_u$ were mapped to "yes" instances of $L_{ne}$, and "no" instances to "no" instances.
- If $L_{ne}$ has an algorithm (is a decidable language), then so does (is) $L_u$.
- We know $L_u$ is not decidable.
- Therefore $L_{ne}$ is not decidable.

9.38

## What about testing for empty language?

Simpler notation: $L_{ne} = \{M \mid L(M) \neq \emptyset\}$.

- (From now on, we stop talking about $f$, the encoding, as much.) Let
$$L_e = \{M \mid L(M) = \emptyset\}.$$

Is $L_e$ r.e.? Is $L_e$ decidable?

- Neither.

- The complement of $L_{ne}$ is

$$L'_{ne} = L_e \cup \{w \mid w \text{ is not the encoding of any Turing machine }\}.$$

- As $\{w \mid w$ is not the encoding of any Turing machine $\}$ is decidable, therefore it is also recursively enumerable.
- Then if $L_e$ is r.e., then so is $L'_{ne}$ (as it is the union of two r.e. languages).
- But if a language and its complement are both r.e., then the language is decidable.
- And we know that $L_{ne}$ is not decidable.
- So $L_e$ cannot be r.e..
- Therefore $L_e$ also cannot be decidable.

9.39

## Turing Machines with an Infinite Language

Infinite language: Let

$$L_\infty = \{M \mid L(M) \text{ is infinite }\}.$$

- Then $L_\infty$ is undecidable:
- Our earlier reduction of $L_u$ to $L_{ne}$ also reduces $L_u$ to $L_\infty$.
- From a candidate instance $(M,w)$ for the universal language, we produce a new machine $M'$ whose language is infinite exactly when $M$ accepts $w$.
- Then "yes" instances of $L_u$ are mapped by our algorithm to "yes" instances of $L_\infty$ and "no" instances of $L_u$ are mapped by our algorithm to "no" instances of $L_\infty$.
- As we cannot decide $L_u$, therefore By Theorem 9.7 we cannot decide $L_\infty$ either.
- The same reduction also shows that $L_A = \{M \mid L(M) = \Sigma^*\}$ is undecidable.

9.40

## Turing Machines with a Finite Language

I claim that $L_{fin} = \{M \mid L(M) \text{ is finite }\}$ is also undecidable.

Proof:

- For a contradiction, suppose that $L_{fin}$ is decidable
- Then its complement, $(L_{fin})'$, is also decidable.
- Writing

$$(L_{fin})' = L_\infty \cup \underbrace{\{w \mid w \text{ is not the encoding of any Turing machine }\}}_{\text{decidable}},$$

  we will have the desired contradiction by applying the following Lemma (as we already know that $L_\infty$ is undecidable).

9.41

## Turing Machines with a Finite Language

Lemma: If sets $A$ and $B$ satisfy $A \cap B = \emptyset$, and if $A \cup B$ and $B$ are both decidable, then $A$ is decidable.

Proof:

- Let $x$ be a candidate for membership in $A$.
- Test $x$ for membership in $A \cup B$.
  - If $x \notin A \cup B$, then reject $x$,
  - otherwise, test $x$ for membership in $B$.
    * If $x \in B$, then reject $x$,
    * otherwise, accept $x$.
- As $A \cap B = \emptyset$, this algorithm will accept $x$ if and only if $x \in A$, as required. □

We then get the desired result on the previous slide by taking

$$\begin{aligned} A &= L_\infty, \text{ and} \\ B &= \{w \mid w \text{ is not the encoding of any Turing machine }\}. \end{aligned}$$

9.42

## Turing Machines with a Non-Regular Language

Turing machines with non-regular languages:

- Let $L_{nreg} = \{M \mid L(M) \text{ is not regular}\}$.
- We will reduce membership in $L_u$ to membership in $L_{nreg}$, then apply Theorem 9.7.
- Given a candidate instance $(M,w)$ for $L_u$, we construct a new machine $M'$ that accepts a non-regular language if $M$ accepts $w$, and a regular language if $M$ does not accept $w$.
- For any input $x$, our new machine $M'$ simulates $M$ on $w$.
  - If $M$ accepts $w$, then $M'$ accepts $x$ if and only if $x = 0^i1^i$, for some $i \geq 0$ (and rejects $x$ otherwise).
  - (We know that the set of all such words is not a regular language.)
  - If $M$ does not accept $w$, then $M'$ rejects $x$.
  - (The empty language is regular by definition.)
- Then
$$L(M') = \begin{cases} \{0^i1^i \mid i \geq 0\} & \text{if } M \text{ accepts } w \\ \emptyset & \text{otherwise} \end{cases}$$
- We have reduced membership in $L_u$ to membership in $L_{nreg}$.
- But $L_u$ is undecidable.
- So by Theorem 9.7, $L_{nreg}$ is undecidable, too.

9.43

## Turing Machines with a Regular Language

I claim that $L_{reg} = \{M \mid L(M) \text{ is regular }\}$ is also undecidable.
Proof:

- For a contradiction, suppose that $L_{reg}$ is decidable
- Then its complement, $(L_{reg})'$, is also decidable.
- Writing
$$(L_{reg})' = L_{nreg} \cup \underbrace{\{w \mid w \text{ is not the encoding of any Turing machine }\}}_{\text{decidable}},$$
we will have the desired contradiction by applying the previous Lemma (as we already know that $L_{nreg}$ is undecidable) with:
$$\begin{aligned} A &= L_{nreg}, \text{ and} \\ B &= \{w \mid w \text{ is not the encoding of any Turing machine }\}. \end{aligned}$$

These results show that any "interesting" property of Turing machine languages is not decidable. We make this notion precise in the following Theorem.

9.44

## Rice's Theorem

Theorem: Let $P$ be a property of some, but not all, recursively enumerable languages. (I.e. $P$ is some non-empty proper subclass of the class of r.e. languages.) Then the language $L_P = \{M | L(M) \in P\}$ is undecidable.
Remark: Here we only work with the identifiers for legal Turing machines.
Proof: Case 1: Suppose that the empty language, $\emptyset$, is not in $P$.

- Let $L$ be a non-empty recursively enumerable language that is in $P$.
- Let $M_L$ be a Turing machine that accepts $L$.
- We will reduce membership in $L_u$ to membership in $L_P$, then apply Theorem 9.7.
- Let $(M,w)$ be a candidate instance for $L_u$.
- Create a new machine, $M'$ that, on any input $x$, simulates $U$ on $(M,w)$.
  - If $U$ accepts $(M,w)$, then we simulate $M_L$ on $x$.
    - If $M_L$ accepts $x$, then $M'$ accepts $x$.
    - If $U$ rejects $(M,w)$ or if $M_L$ rejects $x$, then $M'$ rejects $x$.
    - If $U$ runs forever on $(M,w)$, or if $U$ accepts $(M,w)$ and $M_L$ then runs forever on $x$, then $M'$ runs forever on $x$.

9.45

## Rice's Theorem

- The above construction gives us that the language of $M'$ is

$$L(M') = \begin{cases} L & \text{if } U \text{ accepts } (M,w) \\ \emptyset & \text{otherwise} \end{cases}$$

- Let $w'$ identify $M'$.
- Take $w'$ as our candidate for membership in $L_P$.
- Then "yes" instances for $L_u$ are sent to "yes" instances for $L_P$ (as we chose $L \in P$ and $L \neq \emptyset$), and
- "no" instances of $L_u$ are sent to "no" instances for $L_P$ (as $\emptyset \notin P$).
- We have reduced membership in $L_u$ to membership in $L_P$.
- But since $L_u$ is undecidable, then by Theorem 9.7, $L_P$ is also undecidable.

9.46

## And if $P$ holds for $\emptyset$?

Case 2: Suppose that the empty language, $\emptyset$, is in $P$.

- Consider the property $P'$, which is the negation of property $P$.
- Then since $\emptyset \notin P'$, therefore testing membership in $L_{P'}$ is undecidable, by the proof of Case 1.
- Since every Turing machine accepts a recursively enumerable language, therefore $(L_P)' = L_{P'}$.
- For a contradiction, suppose that $L_P$ is decidable.
- Then $(L_P)' = L_{P'}$ is also decidable.
- But this contradicts the fact that testing membership in $L_{P'}$ is undecidable.
- Therefore $L_P$ must be undecidable.

9.47

## What can we decide?

Is any interesting problem about Turing machines decidable?
Not really.
Exceptions:

- "Does this Turing machine have fewer than $k$ states?"
- "Does this Turing machine ever move the tape head left on any input?"
- Mostly irrelevant.

9.48

## Problems about two machines

There are obvious problems about two languages, too: given $M_1$ and $M_2$,

- is $L(M_1) = L(M_2)$?
- is $L(M_1) \subseteq L(M_2)$?
- is $L(M_1) \cap L(M_2) = \emptyset$, i.e. are the languages disjoint?

These are all undecidable, too.

- Suppose we could decide any of these problems.
- Then suppose we wanted to decide, for an arbitrary machine $M$, whether $M \in L_e$. (Recall: that problem is undecidable.)
- Make new machines $M_2$ that rejects all inputs, and $M_3$ that accepts all inputs.
- If $L(M) = L(M_2)$, then $M \in L_e$ (else $M \notin L_e$).
- If $L(M) \subseteq L(M_2)$, then $M \in L_e$ (else $M \notin L_e$).
- If $L(M) \cap L(M_3) = \emptyset$, then $M \in L_e$ (else $M \notin L_e$).

In all three cases, given a machine that decides the desired equality or containment, we could then decide membership in $L_e$, which is undecidable.

Therefore these problems are undecidable, too.

9.49

# 4 Undecidable problems in automata theory

### Decision problems about CFGs and CFLs

We did not answer these questions, before:

- Given two CFGs $G_1$ and $G_2$, is $L(G_1) \cap L(G_2) = \emptyset$?
- Given a CFG $G$, is it ambiguous?

To answer them (which is to say, to show that they are undecidable), we need to define a new problem.
Post's Correspondence Problem is a funny undecidable game (sort of).
This dates to roughly WWII. 9.50

### Post's Correspondence Problem

- We are given a finite set of "tiles", where each tile contains two strings over a finite alphabet $\Sigma$.
$$(a_1,b_1),(a_2,b_2),\ldots,(a_n,b_n).$$
- We want a non-empty string $x$, where it is possible to join together a sequence of tiles from the set (allowing repetition), and where the concatenation of the $a_i$ strings and the concatenation of the $b_i$ strings are both equal to $x$.
- Huh?

9.51

### Example of PCP

Tiles:
$T_1 : (00,001), T_2 : (11,10)$ and $T_3 : (011,1)$.
Want: string $x$ to obtain from both parts of the tiles.

- (Note: we do not know what $x$ is!)
- Guess $x$. Suppose $x = 001100011$.
- Tile sequence: $(T_1,T_2,T_1,T_3)$.
- Look at the first strings: $00+11+00+011 = 001100011$
- And the second strings: $001+10+001+1 = 001100011$
- These are the same.
- We have exhibited a solution to this instance of PCP.
- There are instances of PCP for which no solution exists. See Example 9.14 on p402 of the text.
- So the question naturally arises: "Is there an algorithm to decide whether any given instance of PCP has a solution or not?"
- In other words, "Is PCP decidable?"

9.52

### PCP is undecidable

Theorem: PCP is not decidable. (In other words, given any instance of PCP, no algorithm exists to determine whether that instance can be solved.)

This Theorem is proved by reducing membership in the universal language to deciding PCP:

- Given an instance $(M,x)$ of the universal language $L_u$.
- Compute a set of tiles, such that if $M$ accepts $x$, it also is a "yes" instance of the PCP, and vice versa.
- See the text for details; it is pretty.

9.53

## So what?

We can apply this fact to prove that two CFG problems are undecidable:

- CFG-intersection:
  - Given: Two grammars $G_1$ and $G_2$.
  - Question: Is $L(G_1) \cap L(G_2) \neq \emptyset$?
- CFG-ambiguous:
  - Given: Grammar $G$.
  - Question: Is $G$ ambiguous?

9.54

## Proof for CFG-intersection

Given any PCP instance $(a_1,b_1),(a_2,b_2),\ldots,(a_n,b_n)$, let $A=\{a_1,\ldots,a_n\}$ and $B=\{b_1,\ldots,b_n\}$.

Let $L(A)$ be all strings we can obtain by concatenating some words from $A$ together, ending with a string of tags $c_i$ to indicate the reversed order of the tiles used.

- Example: suppose we append $a_1,a_3,a_7$. Then $a_1a_3a_7c_7c_3c_1$ needs to be in $L(A)$.
- The reversed string of $c_i$ indicates which tiles were included.

$L(A)$ is context free:

- $G_A : A \to \varepsilon \mid a_1Ac_1 \mid a_2Ac_2 \mid \cdots \mid a_nAc_n$.

And similarly we define $B$, $L(B)$ and $G_B$.

Do $L(A)$ and $L(B)$ have any non-empty words in common?

9.55

## Deciding CFG-intersection decides PCP

If $L(A)$ and $L(B)$ have non-empty words in common, then we can solve the instance of PCP:

- The word in $L(A)$ matches the word in $L(B)$.
- We used the same set of tiles (because they both end with the same string of $c_i$).
- We created the same word before the $c_i$.

If $L(A)$ and $L(B)$ have no non-empty words in common, then we cannot solve the instance of PCP.

So if we could decide CFG-intersection, then we could decide PCP, which is undecidable.

- Therefore CFG-intersection is undecidable.

9.56

## Deciding CFG-ambiguity decides PCP

Consider the grammars $G_A$ for $L(A)$ and $G_B$ for $L(B)$. Construct a new grammar $G_{AB}$ with

- variables $A$, $B$ and $S$ (with $S$ as the start variable),
- productions $S \to A|B$,
- all the productions from $G_A$ and
- all the productions from $G_B$.

With this construction completed, we now have the desired result by the following Theorem.

Theorem: $G_{AB}$ is ambiguous if and only if the instance $(A,B)$ of PCP has a solution.

Remark: This Theorem implies that if we can decide CFG-ambiguity, then we can decide PCP. Since we already know that we cannot decide PCP, therefore we conclude that we cannot decide CFG-ambiguity either.

9.57

### Deciding CFG-ambiguity decides PCP

Proof: ("If")

- Suppose that the indices $i_1, i_2, \ldots, i_m$ are a solution to this instance of PCP.
- Then we have these derivations in $G_{AB}$:

$$S \Rightarrow A \Rightarrow a_{i_1} A c_{i_1} \Rightarrow a_{i_1} a_{i_2} A c_{i_2} c_{i_1} \Rightarrow \cdots \Rightarrow a_{i_1} \cdots a_{i_m} c_{i_m} \cdots c_{i_1}$$

$$S \Rightarrow B \Rightarrow b_{i_1} B c_{i_1} \Rightarrow b_{i_1} b_{i_2} B c_{i_2} c_{i_1} \Rightarrow \cdots \Rightarrow b_{i_1} \cdots b_{i_m} c_{i_m} \cdots c_{i_1}$$

- By assumption, we have that $a_{i_1} \cdots a_{i_m} = b_{i_1} \cdots b_{i_m}$, i.e both derivations yield the same terminal string.
- Since the derivations are distinct by construction, therefore we conclude that $G_{AB}$ is ambiguous.

9.58

### Deciding CFG-ambiguity decides PCP

("Only if")

- Assume that $G_{AB}$ is ambiguous.
- The grammars $G_A$ and $G_B$ are unambiguous, because of the trailing tile markers.
- So the only way that a terminal string can have two different derivations in $G_{AB}$ is if one derivation starts with $S \to A$ and the other starts with $S \to B$.
- The string with two different derivations has a tail $c_{i_m} \cdots c_{i_1}$, for some $m \geq 1$.
- This tail gives a solution to the instance of PCP, because what precedes the tail is $a_{i_1} \cdots a_{i_m}$ in the first derivation and $b_{i_1} \cdots b_{i_m}$ in the second, and by assumption these must be equal.

9.59

### Main ideas of Module 9

- There exist undecidable languages.
- Most interesting languages about the languages of Turing machines are undecidable.
- Post's Correspondence problem: several decision problems about context-free grammars are also undecidable.

9.60

## 5 Wrap-up

### End of the course

This is the end of CS 360.

The biggest ideas in this course are:

- Simple machine models can be used to accept increasingly sophisticated languages.
- Problems can be formally stated, and modelled as languages.
- Computation can be formally modelled by a Turing machine.
- There are problems that cannot be solved by computers.
- Reduction: Solve one problem by changing it into another problem that you already know how to solve.

9.61

### Simple models are useful

DFAs and NFAs

- Model computers with a finite amount of memory
- Can verify whether a word comes from a regular language
- Remarkably useful for text searching (though rarely are full-fledged DFA/NFAs used)
- A surprisingly complicated set of languages is actually regular

Main idea: many different extensions to NFA/DFA do not change the power of the underlying model.

9.62

## Simple models

CFGs:

- Very useful in parsing computer language syntax
- Not good at parsing human languages (where context is crucial)
- Can be used for compression

PDAs:

- Like an NFA, but infinite stack memory
- The most trivial way to augment a finite memory
- Accept exactly CFLs.

9.63

## Turing machines

Like a DFA, but:

- Infinite memory
- Move back and forth in the memory
- Real computation tasks
  - Arithmetic
  - Verifying a number is a perfect square
  - Finding a duplicated string
- Our full idea of what a computer really is

Many equivalent models (non-determinism, etc.); much like real computers, except with infinite memory

9.64

## Church-Turing thesis

Basic philosophical basis for the claim that Turing machines are reasonable models of computation.

"Any algorithmic procedure that can be carried out at all can be carried out by a TM."

- First stated in the 1930s.
- We basically believe it; most anything anyone can propose turns out to be equivalent to a TM.

9.65

## Turing machine languages

- Recursive language: Can be decided by a TM
- Recursively enumerable language: Can be enumerated by a TM, or can be accepted by a TM.
- Pretty much any reasonable problem can be characterized as a language, but not all problems are actually solvable by a TM.
- Unsolvable problem: related language is not decidable.

9.66

## Undecidable languages

We saw examples of languages that are undecidable:

- Does this TM accept this word?
- Does this TM accept a regular language?
- Is the intersection of the languages of these two CFGs empty?
- Is this CFG ambiguous?

9.67

## Reduction, to show a problem cannot be solved

To show $P$ is unsolvable:

- Suppose problem $Q$ is unsolvable.
- Show a way of changing instances of $Q$ into instances of $P$.
- If a solution method for $P$ existed, we could use that solve $Q$.
- But $Q$ is unsolvable. Therefore, no solution method for $P$ exists, either.

9.68

## That's all, folks

Good luck on your exams!

9.69