Computability I

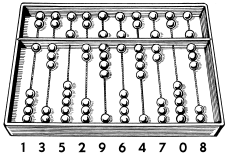
Martin Hoefer

(based on material by Walter Unger)

- Which problems can be solved by a computer?

Computing devices (1)

?



-3000

Pearson Scott Foresman/Wikimedia Commons/Public Domain

1923

Greg Goebel/Wikimedia Commons/Public Domain

1980

1983

2013

Computing devices (2)

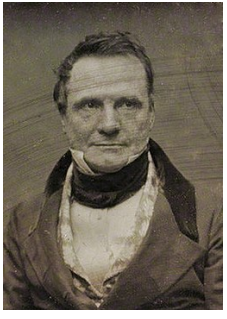
- Central question:
  Which problems can be solved by a computer?

- There are many different computing devices.
  Does the answer depend on the concrete computing device?

Charles Babbage (1791-1871)

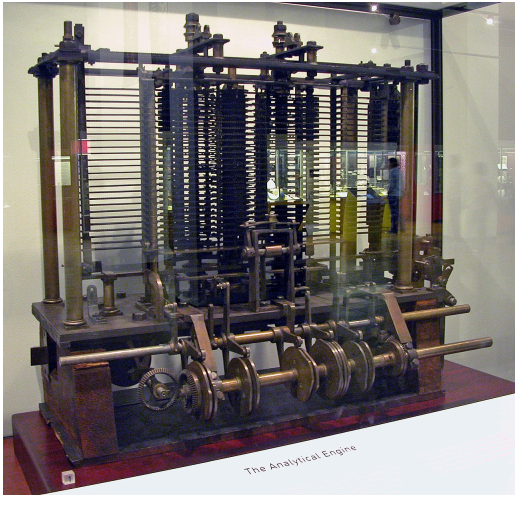
Wikipedia: Charles Babbage was an English polymath.
A mathematician, philosopher, inventor and mechanical engineer, Babbage originated the concept of a digital programmable computer.

Considered by some to be a “father of the computer”, Babbage is credited with inventing the first mechanical computer that eventually led to more complex electronic designs, though all the essential ideas of modern computers are to be found in Babbage’s analytical engine.



The Analytical Engine

Trial model of a part of the Analytical Engine,
built by Babbage, as displayed at the Science Museum (London)



Ability and potential of the computer (1)

Quote (Charles Babbage)

"As soon as an Analytical Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will then arise | by what course of calculation can these results be arrived at by the machine in the shortest time?"

Quote (British Prime Minister Sir Robert Peel, 1842):
"What shall we do to get rid of Mr. Babbage and his calculating machine? Surely if completed it would be worthless as far as science is concerned?"

Ability and potential of the computer (2)

In 1984 the TIME magazine ran a cover story on computer software. The editor of some software magazine was quoted as saying:

"Put the right kind of software into a computer,
and it will do whatever you want it to.
There may be limits on what you can do with the machines themselves, but there are no limits on what you can do with software."

Ability and potential of the computer (2)

In 1984 the TIME magazine ran a cover story on computer software. The editor of some software magazine was quoted as saying:

"Put the right kind of software into a computer,
and it will do whatever you want it to.
There may be limits on what you can do with the machines themselves, but there are no limits on what you can do with software."

That's a **bold** statement.

Ability and potential of the computer (2)

In 1984 the TIME magazine ran a cover story on computer software. The editor of some software magazine was quoted as saying:

"Put the right kind of software into a computer,
and it will do whatever you want it to.
There may be limits on what you can do with the machines themselves, but there are no limits on what you can do with software."

That's a **bold** statement.
And it is wrong. Wrong. Totally wrong. Absolutely wrong.

Ability and potential of the computer (3)

Question

Does there exist a C++ program P,
that outputs "`Hello, World!`" and then stops?

Ability and potential of the computer (3)

Question

Does there exist a C++ program P,
that outputs "`Hello, World!`" and then stops?

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Hello, World!";
    return 0;
}
```

Ability and potential of the computer (4)

Follow-up question

Are you able to analyze a text-file with some C++ program P in the following way?

- If P outputs "`Hello, World!`" and then stops, then you should say "`Good`".
- If P does behave in any other way, then you should say "`Bad`".

Ability and potential of the computer (5)

Follow-up question

Are you able to write a C++ program Q,
that takes as input a file with some C++ program P,
and behaves as follows:

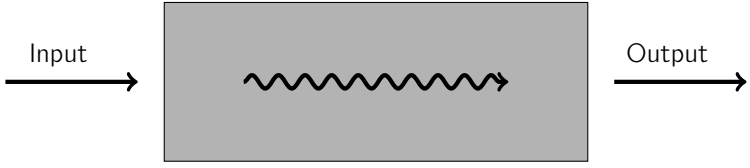
- If P outputs "`Hello, World!`" and then stops,
then program Q outputs "`Good`" and stops.
- If P does behave in any other way,
then program Q outputs "`Bad`" and stops.

The absolute limits of the computer (1)

A computer takes an input, computes, and generates an output:

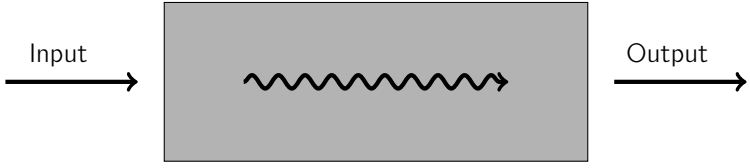
The absolute limits of the computer (1)

A computer takes an input, computes, and generates an output:



The absolute limits of the computer (1)

A computer takes an input, computes, and generates an output:

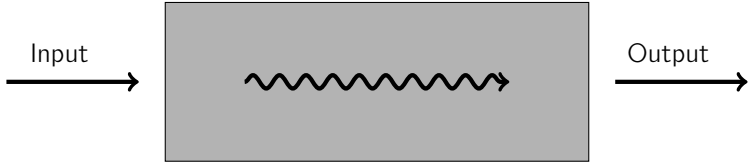


Central question

Do there exist problems,
that cannot be solved by a computer?

The absolute limits of the computer (1)

A computer takes an input, computes, and generates an output:



Central question

Do there exist problems,
that cannot be solved by a computer?

Central question (better formulation)

Do there exist "algorithmic problems" (or "computational problems"),
that cannot be solved by a computer?

The absolute limits of the computer (2)

Central question (better formulation)

Do there exist “algorithmic problems” (or “computational problems”), that cannot be solved by a computer?

The absolute limits of the computer (2)

Central question (better formulation)

Do there exist "algorithmic problems" (or "computational problems"), that cannot be solved by a computer?

The answer

There is no computer program / no algorithm that decides, whether a given computer program ever reaches a given state.

The absolute limits of the computer (2)

Central question (better formulation)

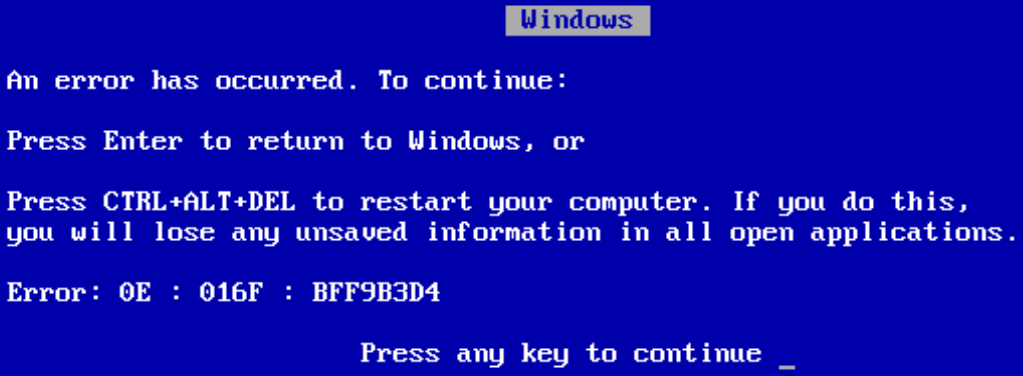
Do there exist "algorithmic problems" (or "computational problems"), that cannot be solved by a computer?

The answer

There is no computer program / no algorithm that decides, whether a given computer program ever reaches a given state.

Example

```
Error: 0E : 016F : BFF9B3D4
```


Windows
An error has occurred. To continue:
Press Enter to return to Windows, or
Press CTRL+ALT+DEL to restart your computer. If you do this,
you will lose any unsaved information in all open applications.
Error: 0E : 016F : BFF9B3D4
Press any key to continue _

The absolute limits of the computer (3)

Once again:

The answer

There is no computer program / no algorithm that decides,
whether a given computer program ever reaches a given state.

(The proof will be discussed later on.)

The absolute limits of the computer (3)

Once again:

The answer

There is no computer program / no algorithm that decides,
whether a given computer program ever reaches a given state.

(The proof will be discussed later on.)

Even worse:

- There are no algorithms for verifying the correctness of a program
- There is no algorithm, that can decide whether a given program always computes and outputs the sum of two input numbers

Alphabets and words

- Inputs and outputs are words over some alphabet Σ
- Examples for alphabets: $\Sigma = \{0, 1, \#\}$
  $\Sigma = \{a, b, c, \ldots, z\}$
- Σ^k is the set of all words of length k, as for instance
  $\{0,1\}^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$
- The **empty word** (the word of length 0) is denoted by ϵ.
  Hence: $\Sigma^0 = \{\epsilon\}$
- $\Sigma^* = \bigcup_{k \in \mathbb{N}_0} \Sigma^k$ is the **Kleene closure** of Σ and contains all words over Σ.
- The words in Σ^* can be enumerated by increasing length:

  $\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, \ldots$

Computational problems (1)

- A computational problem translates an input into an output
- Hence, the problem corresponds to a function $f : \Sigma^* \to \Sigma'^*$
- For input $x \in \Sigma^*$, the corresponding output is $f(x) \in \Sigma'^*$

Computational problems (1)

- A computational problem translates an input into an output
- Hence, the problem corresponds to a function $f : \Sigma^* \to \Sigma'^*$
- For input $x \in \Sigma^*$, the corresponding output is $f(x) \in \Sigma'^*$

Example: Multiplication

For two given natural numbers $i_1, i_2 \in \mathbb{N}$ (input), we would like to find the corresponding product $i_1 \cdot i_2$ (output).

The numbers i_1 and i_2 are separated by a separation symbol $\#$, and the alphabet is $\Sigma = \{0, 1, \#\}$.

The corresponding function $f : \Sigma^* \to \Sigma^*$ is given by

$$f(\mathrm{bin}(i_1)\#\,\mathrm{bin}(i_2)) \;=\; \mathrm{bin}(i_1 \cdot i_2)$$

Computational problems (2)

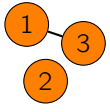
- Many problems can be formulated as Yes-No-questions.
- Such problems are called **decision problems**.
  Their corresponding functions are of the form $f : \Sigma^* \to \{0, 1\}$, where 0 means "No" and 1 means "Yes".
- Let $L = f^{-1}(1) \subseteq \Sigma^*$ be the set of all inputs with answer "Yes".
  Then L is a language over alphabet Σ.
- We say: Language L is the language that corresponds to the underlying decision problem.

Example: Decision problems as languages

Example: Graph connectivity

Problem: For a given undirected graph $G = (V, E)$ we want to determine whether G is connected.

The graph G is appropriately encoded as a word $\text{code}(G) \in \Sigma^*$, for instance as an adjacency matrix encoded in binary.

Example: 

$\rightsquigarrow$ $\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$ $\rightsquigarrow$ 001000100

The language that corresponds to the decision problem is

$$L = \{ w \in \Sigma^* \mid \exists \text{ graph } G\text{: } w = \text{code}(G), \text{ and } G \text{ is connected} \}$$

Definition

Definition: Countable set

A set M is **countable**,
if either M is empty,
or if there exists a surjective function $c : \mathbb{N} \to M$.

Definition

Definition: Countable set

A set M is **countable**,
if either M is empty,
or if there exists a surjective function $c : \mathbb{N} \to M$.

- Every finite set M is countable
- The elements of a countable set can be numbered and enumerated
- For a **countably infinite** set M, there always exists a bijective (bijective = surjective+injective) function $c : \mathbb{N} \to M$: Repeated elements in may simply be skipped in the enumeration
- Hence: Countably infinite sets have the **same** cardinality as set $\mathbb{N}$

Examples of countable sets (1)

Example: $\mathbb{Z}$

The set $\mathbb{Z}$ of integers is countable. The function

$$c(i) = \begin{cases} i/2 & \text{for even } i \\ -(i+1)/2 & \text{for odd } i \end{cases}$$

yields a bijection as desired. The resulting enumeration is:

$$0,\ -1, 1,\ -2, 2,\ -3, 3,\ -4, 4,\ -5, 5,\ \ldots,$$

Examples of countable sets (2a)

Example: $\mathbb{Q}$

The set $\mathbb{Q}$ of rational numbers is countable. One possible enumeration is:

$$0,\ \frac{1}{1},\ \frac{2}{1},\frac{1}{2},\ \frac{3}{1},\frac{2}{2},\frac{1}{3},\ \ldots,\ \frac{i}{1},\frac{i-1}{2},\frac{i-2}{3},\ldots\frac{1}{i},\ \ldots$$

Examples of countable sets (2b)

	1	2	3	4	5	6	$\cdots$
1	1/1	2/1	3/1	4/1	5/1	6/1	
2	1/2	2/2	3/2	4/2	5/2	6/2	
3	1/3	2/3	3/3	4/3	5/3	6/3	$\cdots$
4	1/4	2/4	3/4	4/4	5/4	6/4	
5	1/5	2/5	3/5	4/5	5/5	6/5	
6	1/6	2/6	3/6	4/6	5/6	6/6	
$\vdots$				$\vdots$			$\ddots$

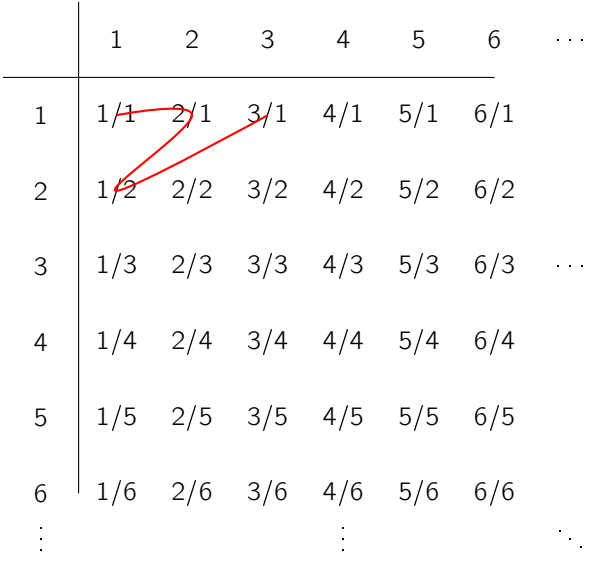
Examples of countable sets (2b)

	1	2	3	4	5	6	⋯
1	1/1	2/1	3/1	4/1	5/1	6/1	
2	1/2	2/2	3/2	4/2	5/2	6/2	
3	1/3	2/3	3/3	4/3	5/3	6/3	⋯
4	1/4	2/4	3/4	4/4	5/4	6/4	
5	1/5	2/5	3/5	4/5	5/5	6/5	
6	1/6	2/6	3/6	4/6	5/6	6/6	
⋮				⋮			⋱

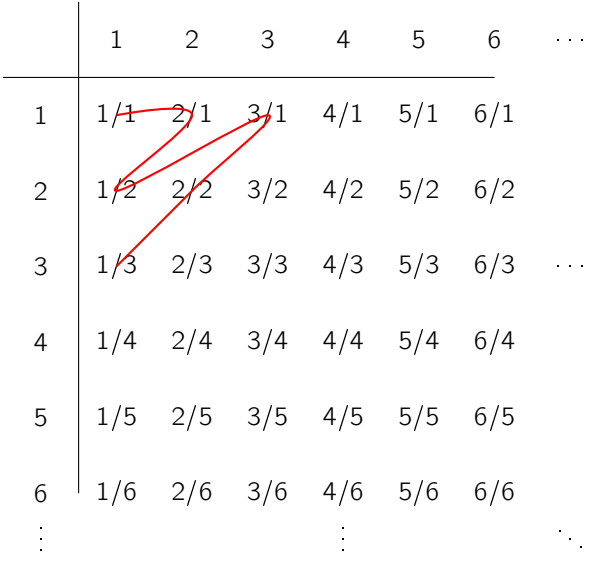
Examples of countable sets (2b)

	1	2	3	4	5	6	⋯
1	1/1	2/1	3/1	4/1	5/1	6/1	
2	1/2	2/2	3/2	4/2	5/2	6/2	
3	1/3	2/3	3/3	4/3	5/3	6/3	⋯
4	1/4	2/4	3/4	4/4	5/4	6/4	
5	1/5	2/5	3/5	4/5	5/5	6/5	
6	1/6	2/6	3/6	4/6	5/6	6/6	
⋮				⋮			⋱

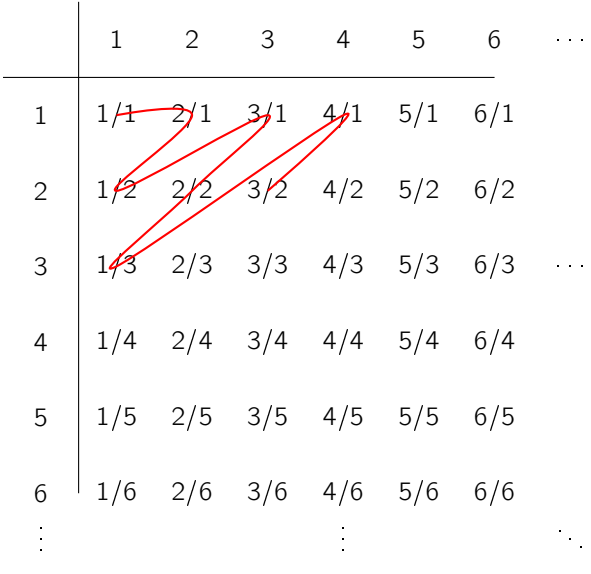
Examples of countable sets (2b)



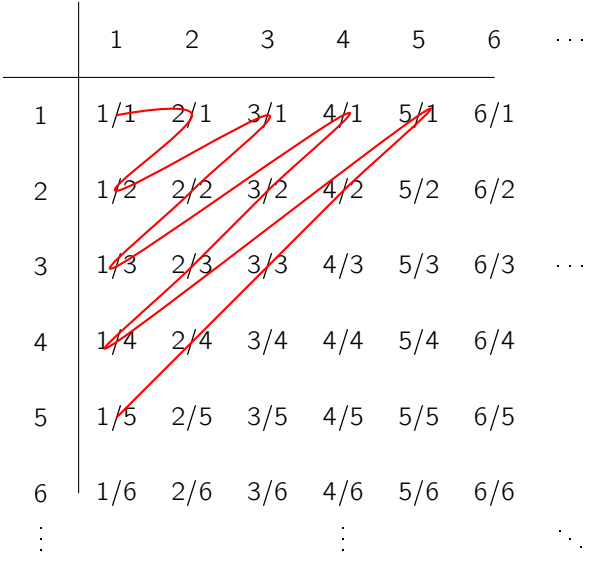
Examples of countable sets (2b)



Examples of countable sets (2b)



Examples of countable sets (2b)



Examples of countable sets (3)

Example: Σ^*

The set Σ^* of words over a finite alphabet Σ is countable.
For instance, $\{0,1\}^*$ may be enumerated in **canonical order** as

$$\epsilon,\ 0, 1,\ 00, 01, 10, 11,\ 000, 001, 010, 011, 100, 101, 110, 111,\ \ldots$$

Notation (will be used in later lectures)

The i-th word over the binary alphabet $\Sigma = \{0,1\}$ in the canonical order will be denoted by w_i.

- There exists a C++ program that
  takes as input an integer i and outputs w_i.

Examples of countable sets (4)

Example: C++ programs

- The set of words $\langle P \rangle$ that describe valid C++ programs is countable.
- The set of valid C++ programs is countable.

Notation (will be used in later lectures)

The i-th C++ program in the canonical ordering of the words $\langle P \rangle$ will be denoted by P_i.

- There exists a C++ program that
  takes as input an integer i and outputs P_i.

Definition

Definition: Uncountable set

A set M is **uncountable**, if it is not countable.

Uncountability proof (1)

The **power set** $\mathcal{P}(\mathbb{N})$ is the set of all subsets of $\mathbb{N} = \{1, 2, 3, \ldots\}$.

Theorem

The set $\mathcal{P}(\mathbb{N})$ is uncountable.

Uncountability proof (1)

The **power set** $\mathcal{P}(\mathbb{N})$ is the set of all subsets of $\mathbb{N} = \{1, 2, 3, \ldots\}$.

Theorem

The set $\mathcal{P}(\mathbb{N})$ is uncountable.

Proof: by diagonalization

- Let us assume for the sake of contradiction that $\mathcal{P}(\mathbb{N})$ is countable
- Let $S_0, S_1, S_2, S_3, \ldots$ be an enumeration of $\mathcal{P}(\mathbb{N})$.
- We define a 2-dimensional infinite matrix $(A_{i,j})_{i,j \in \mathbb{N}}$ with

$$A_{i,j} = \begin{cases} 1 & \text{if } j \in S_i \\ 0 & \text{otherwise} \end{cases}$$

Uncountability proof (2)

This matrix A might for instance look as follows:

	0	1	2	3	4	5	6	
S_0	**0**	1	1	0	1	0	1	$\cdots$
S_1	1	**1**	1	0	1	0	1	$\cdots$
S_2	0	0	**1**	0	1	0	1	$\cdots$
S_3	0	1	1	**0**	0	0	1	$\cdots$
S_4	0	1	0	0	**1**	0	1	$\cdots$
S_5	0	1	1	0	1	**0**	0	$\cdots$
S_6	1	1	1	0	1	0	**0**	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

Uncountability proof (2)

This matrix A might for instance look as follows:

$\{1, 2, 4, 6, \ldots\} =$

	0	1	2	3	4	5	6	
S_0	**0**	1	1	0	1	0	1	$\cdots$
S_1	1	**1**	1	0	1	0	1	$\cdots$
S_2	0	0	**1**	0	1	0	1	$\cdots$
S_3	0	1	1	**0**	0	0	1	$\cdots$
S_4	0	1	0	0	**1**	0	1	$\cdots$
S_5	0	1	1	0	1	**0**	0	$\cdots$
S_6	1	1	1	0	1	0	**0**	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

Uncountability proof (2)

This matrix A might for instance look as follows:

		0	1	2	3	4	5	6	
$\{1,2,4,6,\ldots\} =$	S_0	**0**	1	1	0	1	0	1	$\cdots$
$\{0,1,2,4,6,\ldots\} =$	S_1	1	**1**	1	0	1	0	1	$\cdots$
	S_2	0	0	**1**	0	1	0	1	$\cdots$
	S_3	0	1	1	**0**	0	0	1	$\cdots$
	S_4	0	1	0	0	**1**	0	1	$\cdots$
	S_5	0	1	1	0	1	**0**	0	$\cdots$
	S_6	1	1	1	0	1	0	**0**	$\cdots$
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

Uncountability proof (2)

This matrix A might for instance look as follows:

		0	1	2	3	4	5	6	
$\{1,2,4,6,\ldots\} =$	S_0	**0**	1	1	0	1	0	1	$\cdots$
$\{0,1,2,4,6,\ldots\} =$	S_1	1	**1**	1	0	1	0	1	$\cdots$
	S_2	0	0	**1**	0	1	0	1	$\cdots$
	S_3	0	1	1	**0**	0	0	1	$\cdots$
	S_4	0	1	0	0	**1**	0	1	$\cdots$
	S_5	0	1	1	0	1	**0**	0	$\cdots$
	S_6	1	1	1	0	1	0	**0**	$\cdots$
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

- We define the diagonal set $S_{\text{diag}} = \{i \in \mathbb{N} \mid A_{i,i} = 1\}$

Uncountability proof (2)

This matrix A might for instance look as follows:

		0	1	2	3	4	5	6	
$\{1,2,4,6,\ldots\} =$	S_0	**0**	1	1	0	1	0	1	$\cdots$
$\{0,1,2,4,6,\ldots\} =$	S_1	1	**1**	1	0	1	0	1	$\cdots$
	S_2	0	0	**1**	0	1	0	1	$\cdots$
	S_3	0	1	1	**0**	0	0	1	$\cdots$
	S_4	0	1	0	0	**1**	0	1	$\cdots$
	S_5	0	1	1	0	1	**0**	0	$\cdots$
	S_6	1	1	1	0	1	0	**0**	$\cdots$
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

- We define the diagonal set $S_{\text{diag}} = \{i \in \mathbb{N} \mid A_{i,i} = 1\}$
- The complement of the diagonal set is $\overline{S}_{\text{diag}} = \mathbb{N} \setminus S_{\text{diag}} = \{i \in \mathbb{N} \mid A_{i,i} = 0\}$

Uncountability proof (3)

- Note: The set $\overline{S}_{\text{diag}}$ is a subset of $\mathbb{N}$

Uncountability proof (3)

- Note: The set $\overline{S}_{\text{diag}}$ is a subset of $\mathbb{N}$
- Hence: $\overline{S}_{\text{diag}}$ shows up in the enumeration $S_1, S_2, \ldots$ of $\mathcal{P}(\mathbb{N})$
- Hence: There exists $k \in \mathbb{N}$ with $\overline{S}_{\text{diag}} = S_k$.
- What is the value of the matrix element $A_{k,k}$?

Uncountability proof (3)

- Note: The set $\overline{S}_{\text{diag}}$ is a subset of $\mathbb{N}$
- Hence: $\overline{S}_{\text{diag}}$ shows up in the enumeration $S_1, S_2, \ldots$ of $\mathcal{P}(\mathbb{N})$
- Hence: There exists $k \in \mathbb{N}$ with $\overline{S}_{\text{diag}} = S_k$.
- What is the value of the matrix element $A_{k,k}$?

- $A_{k,k} = 1 \overset{\text{Def. } \overline{S}_{\text{diag}}}{\Rightarrow} k \notin \overline{S}_{\text{diag}} \Rightarrow k \notin S_k \overset{\text{Def. } A}{\Rightarrow} A_{k,k} = 0$

Uncountability proof (3)

- Note: The set $\overline{S}_{\text{diag}}$ is a subset of $\mathbb{N}$
- Hence: $\overline{S}_{\text{diag}}$ shows up in the enumeration $S_1, S_2, \ldots$ of $\mathcal{P}(\mathbb{N})$
- Hence: There exists $k \in \mathbb{N}$ with $\overline{S}_{\text{diag}} = S_k$.
- What is the value of the matrix element $A_{k,k}$?

- $A_{k,k} = 1 \overset{\text{Def. } \overline{S}_{\text{diag}}}{\Rightarrow} k \notin \overline{S}_{\text{diag}} \Rightarrow k \notin S_k \overset{\text{Def. } A}{\Rightarrow} A_{k,k} = 0$
- $A_{k,k} = 0 \overset{\text{Def. } \overline{S}_{\text{diag}}}{\Rightarrow} k \in \overline{S}_{\text{diag}} \Rightarrow k \in S_k \overset{\text{Def. } A}{\Rightarrow} A_{k,k} = 1$

Uncountability proof (3)

- Note: The set $\overline{S}_{\text{diag}}$ is a subset of $\mathbb{N}$
- Hence: $\overline{S}_{\text{diag}}$ shows up in the enumeration $S_1, S_2, \ldots$ of $\mathcal{P}(\mathbb{N})$
- Hence: There exists $k \in \mathbb{N}$ with $\overline{S}_{\text{diag}} = S_k$.
- What is the value of the matrix element $A_{k,k}$?

- $A_{k,k} = 1 \overset{\text{Def. } \overline{S}_{\text{diag}}}{\Rightarrow} k \notin \overline{S}_{\text{diag}} \Rightarrow k \notin S_k \overset{\text{Def. } A}{\Rightarrow} A_{k,k} = 0$
- $A_{k,k} = 0 \overset{\text{Def. } \overline{S}_{\text{diag}}}{\Rightarrow} k \in \overline{S}_{\text{diag}} \Rightarrow k \in S_k \overset{\text{Def. } A}{\Rightarrow} A_{k,k} = 1$

- The case $A_{k,k} = 1$ yields a contradiction, and also the case $A_{k,k} = 0$ yields a contradiction. Hence there is no enumeration of $\mathcal{P}(\mathbb{N})$.
  This completes our proof!

Test: Which set has the larger cardinality?

1. $\{1, \ldots, n\}$ $\mathbb{N}$

Test: Which set has the larger cardinality?

1. $\{1, \ldots, n\}$ $<$ $\mathbb{N}$

Test: Which set has the larger cardinality?

1. $\{1, \ldots, n\}$ (finite) $<$ $\mathbb{N}$ (countable)

Test: Which set has the larger cardinality?

1. $\{1, \ldots, n\}$ (finite) $<$ $\mathbb{N}$ (countable)
2. $\{\☺\}^*$ $\quad$ $\mathcal{P}(\mathbb{N})$

Test: Which set has the larger cardinality?

1. $\{1, \ldots, n\}$ (finite) $<$ $\mathbb{N}$ (countable)
2. $\{\text{☺}\}^*$ $<$ $\mathcal{P}(\mathbb{N})$

Test: Which set has the larger cardinality?

1. $\{1, \ldots, n\}$ (finite) $<$ $\mathbb{N}$ (countable)
2. $\{☺\}^*$ (countable) $<$ $\mathcal{P}(\mathbb{N})$ (uncountable)

Test: Which set has the larger cardinality?

1.	$\{1, \ldots, n\}$ finite	$<$	$\mathbb{N}$ countable
2.	$\{\text{☺}\}^*$ countable	$<$	$\mathcal{P}(\mathbb{N})$ uncountable
3.	$\mathbb{R}$		$\mathcal{P}(\mathbb{N})$

Test: Which set has the larger cardinality?

1.	$\{1, \ldots, n\}$ finite	$<$	$\mathbb{N}$ countable
2.	$\{\text{☺}\}^*$ countable	$<$	$\mathcal{P}(\mathbb{N})$ uncountable
3.	$\mathbb{R}$	$=$	$\mathcal{P}(\mathbb{N})$

Test: Which set has the larger cardinality?

1.	$\{1, \ldots, n\}$ finite	$<$	$\mathbb{N}$ countable
2.	$\{\text{☺}\}^*$ countable	$<$	$\mathcal{P}(\mathbb{N})$ uncountable
3.	$\mathbb{R}$ uncountable	$=$	$\mathcal{P}(\mathbb{N})$ uncountable

Test: Which set has the larger cardinality?

1.	$\{1, \ldots, n\}$ finite	$<$	$\mathbb{N}$ countable
2.	$\{\☺\}^*$ countable	$<$	$\mathcal{P}(\mathbb{N})$ uncountable
3.	$\mathbb{R}$ uncountable	$=$	$\mathcal{P}(\mathbb{N})$ uncountable
4.	Graphs with $\exists n : V(G) = \{1, \ldots, n\}$		$\{0,1\}^*$

Test: Which set has the larger cardinality?

1.	$\{1, \ldots, n\}$ finite	$<$	$\mathbb{N}$ countable
2.	$\{\☺\}^*$ countable	$<$	$\mathcal{P}(\mathbb{N})$ uncountable
3.	$\mathbb{R}$ uncountable	$=$	$\mathcal{P}(\mathbb{N})$ uncountable
4.	Graphs with $\exists n : V(G) = \{1, \ldots, n\}$	$=$	$\{0,1\}^*$

Test: Which set has the larger cardinality?

1.	$\{1, \ldots, n\}$ finite	$<$	$\mathbb{N}$ countable
2.	$\{\☺\}^*$ countable	$<$	$\mathcal{P}(\mathbb{N})$ uncountable
3.	$\mathbb{R}$ uncountable	$=$	$\mathcal{P}(\mathbb{N})$ uncountable
4.	Graphs with $\exists n : V(G) = \{1, \ldots, n\}$ countable	$=$	$\{0,1\}^*$ countable

Test: Which set has the larger cardinality?

1.	$\{1, \ldots, n\}$ finite	$<$	$\mathbb{N}$ countable
2.	$\{\☺\}^*$ countable	$<$	$\mathcal{P}(\mathbb{N})$ uncountable
3.	$\mathbb{R}$ uncountable	$=$	$\mathcal{P}(\mathbb{N})$ uncountable
4.	Graphs with $\exists n : V(G) = \{1, \ldots, n\}$ countable	$=$	$\{0,1\}^*$ countable
5.	$\mathbb{N}$		$\{\mathbb{R}\}$

Test: Which set has the larger cardinality?

1.	$\{1, \ldots, n\}$ finite	$<$	$\mathbb{N}$ countable
2.	$\{\☺\}^*$ countable	$<$	$\mathcal{P}(\mathbb{N})$ uncountable
3.	$\mathbb{R}$ uncountable	$=$	$\mathcal{P}(\mathbb{N})$ uncountable
4.	Graphs with $\exists n : V(G) = \{1, \ldots, n\}$ countable	$=$	$\{0,1\}^*$ countable
5.	$\mathbb{N}$	$>$	$\{\mathbb{R}\}$

Test: Which set has the larger cardinality?

1.	$\{1, \ldots, n\}$ finite	$<$	$\mathbb{N}$ countable
2.	$\{\☺\}^*$ countable	$<$	$\mathcal{P}(\mathbb{N})$ uncountable
3.	$\mathbb{R}$ uncountable	$=$	$\mathcal{P}(\mathbb{N})$ uncountable
4.	Graphs with $\exists n : V(G) = \{1, \ldots, n\}$ countable	$=$	$\{0,1\}^*$ countable
5.	$\mathbb{N}$ countable	$>$	$\{\mathbb{R}\}$ one-element

Test: Which set has the larger cardinality?

1.	$\{1, \ldots, n\}$ finite	$<$	$\mathbb{N}$ countable
2.	$\{\text{☺}\}^*$ countable	$<$	$\mathcal{P}(\mathbb{N})$ uncountable
3.	$\mathbb{R}$ uncountable	$=$	$\mathcal{P}(\mathbb{N})$ uncountable
4.	Graphs with $\exists n : V(G) = \{1, \ldots, n\}$ countable	$=$	$\{0,1\}^*$ countable
5.	$\mathbb{N}$ countable	$>$	$\{\mathbb{R}\}$ one-element
6.	C++ Programs		$\mathcal{P}(\{0,1\}^*)$

Test: Which set has the larger cardinality?

1.	$\{1, \ldots, n\}$ finite	$<$	$\mathbb{N}$ countable
2.	$\{\☺\}^*$ countable	$<$	$\mathcal{P}(\mathbb{N})$ uncountable
3.	$\mathbb{R}$ uncountable	$=$	$\mathcal{P}(\mathbb{N})$ uncountable
4.	Graphs with $\exists n : V(G) = \{1, \ldots, n\}$ countable	$=$	$\{0,1\}^*$ countable
5.	$\mathbb{N}$ countable	$>$	$\{\mathbb{R}\}$ one-element
6.	C++ Programs	$<$	$\mathcal{P}(\{0,1\}^*)$

Test: Which set has the larger cardinality?

1.	$\{1, \ldots, n\}$ finite	$<$	$\mathbb{N}$ countable
2.	$\{\☺\}^*$ countable	$<$	$\mathcal{P}(\mathbb{N})$ uncountable
3.	$\mathbb{R}$ uncountable	$=$	$\mathcal{P}(\mathbb{N})$ uncountable
4.	Graphs with $\exists n : V(G) = \{1, \ldots, n\}$ countable	$=$	$\{0,1\}^*$ countable
5.	$\mathbb{N}$ countable	$>$	$\{\mathbb{R}\}$ one-element
6.	C++ Programs countable	$<$	$\mathcal{P}(\{0,1\}^*)$ uncountable

Crucial question

Crucial question

Which functions can be computed by an algorithm?
Which languages can be decided by an algorithm?

Crucial question

Crucial question

Which functions can be computed by an algorithm?
Which languages can be decided by an algorithm?

- The question is not clearly formulated.
  For an answer, we need to fix the definition of an algorithm.

Crucial question

Crucial question

Which functions can be computed by an algorithm?
Which languages can be decided by an algorithm?

- The question is not clearly formulated.
  For an answer, we need to fix the definition of an algorithm.
- Standard definition in most theory courses: Turing machine (TM) or Random Access Machine (RAM)

Crucial question

Crucial question

Which functions can be computed by an algorithm?
Which languages can be decided by an algorithm?

- The question is not clearly formulated.
  For an answer, we need to fix the definition of an algorithm.
- Standard definition in most theory courses: Turing machine (TM) or Random Access Machine (RAM)
- Our definition in this course: **C++ program**
  Note: JAVA is fine, too.
  Note: Haskell is fine, too.
  Note: Every (higher) programming language would be fine.

The Church-Turing thesis (1)

Church-Turing thesis

The class of "intuitively computable" functions coincides with the class of functions that can be computed by a Turing machine.

The Church-Turing thesis (1)

Church-Turing thesis

The class of "intuitively computable" functions coincides with the class of functions that can be computed by a Turing machine.

Church-Turing thesis (our variant as used in this course)

The class of "intuitively computable" functions coincides with the class of functions that can be computed by a **C++ program**.

The Church-Turing thesis (1)

Church-Turing thesis

The class of "intuitively computable" functions coincides with the class of functions that can be computed by a Turing machine.

Church-Turing thesis (our variant as used in this course)

The class of "intuitively computable" functions coincides with the class of functions that can be computed by a **C++ program**.

All known "reasonable" models of computation are exactly as powerful as the Turing machine:

- Lambda-calculus by Alonzo Church: equivalent to Turing machine
- Canonical systems by Emil Post: equivalent to Turing machine
- μ-recursive functions by Kurt Gödel: equivalent to Turing machine

The Church-Turing thesis (2)

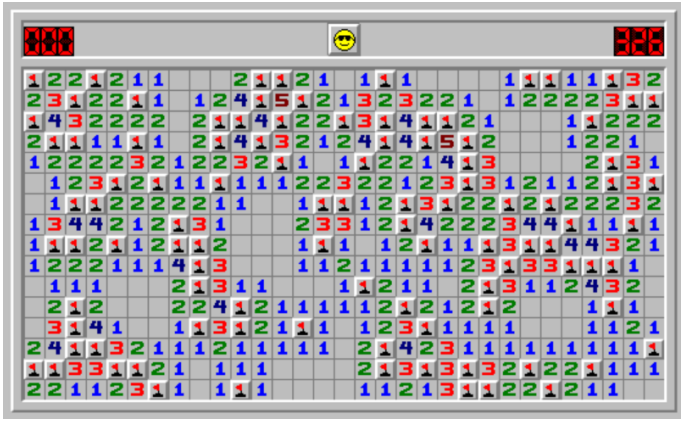
- All standard higher programming languages are exactly as powerful as Turing machines and C++ programs: Algol, Pascal, C, FORTRAN, COBOL, Java, Smalltalk, Ada, C++, Python, LISP, Haskell, PROLOG, etc.
- PostScript, Tex, Latex are exactly as powerful as Turing machines
- Even PowerPoint is exactly as powerful as the Turing machine (because of its Animated Features)

The Church-Turing thesis (2)

- All standard higher programming languages are exactly as powerful as Turing machines and C++ programs: Algol, Pascal, C, FORTRAN, COBOL, Java, Smalltalk, Ada, C++, Python, LISP, Haskell, PROLOG, etc.
- PostScript, Tex, Latex are exactly as powerful as Turing machines
- Even PowerPoint is exactly as powerful as the Turing machine (because of its Animated Features)

- Pure HTML (without JavaScript; without browser) is **strictly weaker** than the Turing machine
- Table Calculations (without loops) are **strictly weaker** than the Turing machine

The Church-Turing thesis (3)



Fun fact: Certain infinite variants of the MineSweeper game are exactly as powerful as Turing machines and C++ programs.
(Richard Kayes: "Infinite Versions of Minesweeper are Turing-complete", 2007)

Alonzo Church (1903–1995)

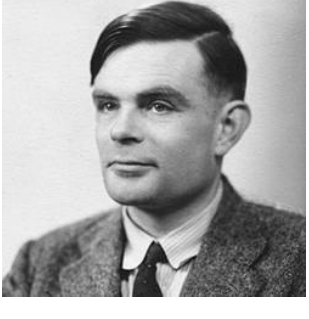
Wikipedia: Alonzo Church was an American mathematician and logician who made major contributions to mathematical logic and the foundations of theoretical computer science.

He is best known for the lambda calculus, Church-Turing thesis, proving the undecidability of the Entscheidungsproblem, Frege-Church ontology, and the Church-Rosser theorem.

Alan Mathison Turing OBE FRS (1912–1954)

Wikipedia: Alan Turing was an English computer scientist, mathematician, logician, cryptanalyst, philosopher and theoretical biologist.

Turing was highly influential in the development of theoretical computer science, providing a formalisation of the concepts of algorithm and computation with the Turing machine, which can be considered a model of a general purpose computer. Turing is widely considered to be the father of theoretical computer science and artificial intelligence.



Computable and recognizable languages (1)

Definition

A function $f : \Sigma^* \to \Sigma^*$ is **Turing-computable**,
if there exists a C++ program that on every input x terminates
and outputs the function value $f(x)$.

- Remark: "**Turing-computable**" is often simply called "**computable**"

Computable and recognizable languages (1)

Definition

A function $f : \Sigma^* \to \Sigma^*$ is **Turing-computable**,
if there exists a C++ program that on every input x terminates and outputs the function value $f(x)$.

- Remark: "**Turing-computable**" is often simply called "**computable**"

Definition

A language $L \subseteq \Sigma^*$ is **Turing-decidable**,
if there exists a C++ program that terminates on all inputs, and that accepts input w if and only if $w \in L$.

- Remark: "**Turing-decidable**" is often simply called "**decidable**"
- Remark: "accept" $\Leftrightarrow$ output 1, and "reject" $\Leftrightarrow$ output 0

Computable and recognizable languages (2)

Definition

A language $L \subseteq \Sigma^*$ is **Turing-recognizable**,
if there exists a C++ program that
for all $w \in L$: terminates and accepts, and
for all $w \notin L$: either does not terminate, or does terminate and reject.

Computable and recognizable languages (2)

Definition

A language $L \subseteq \Sigma^*$ is **Turing-recognizable**,
if there exists a C++ program that
for all $w \in L$: terminates and accepts, and
for all $w \notin L$: either does not terminate, or does terminate and reject.

Equivalent definition

A language $L \subseteq \Sigma^*$ is **Turing-recognizable**,
if there exists a C++ program that
for all $w \in L$: does terminate, and
for all $w \notin L$: does not terminate.

Computable and recognizable languages (3)

True or false?

If $L \subseteq \Sigma^*$ is Turing-decidable,
then L is also Turing-recognizable.

True or false?

If $L \subseteq \Sigma^*$ is Turing-recognizable,
then L is also Turing-decidable.

C++ programs (1)

A word w is **accepted** by program P, if on input w the program eventually terminates and outputs YES.

Notation

For a C++ program P,
we let $L(P)$ denote the set of all words that are accepted by P.

C++ programs (1)

A word w is **accepted** by program P, if on input w the program eventually terminates and outputs YES.

Notation

For a C++ program P,
we let $L(P)$ denote the set of all words that are accepted by P.

True or false?

$L \subseteq \Sigma^*$ is Turing-recognizable, if and only if
there exists a C++ program P with $L(P) = L$.

True or false?

$L \subseteq \Sigma^*$ is Turing-decidable, if and only if
there exists a C++ program P with $L(P) = L$.

C++ programs (2)

- Every C++ program is a piece of text (sequence of ascii symbols).
  Every C++ program is a word over the alphabet of ascii symbols.

C++ programs (2)

- Every C++ program is a piece of text (sequence of ascii symbols).
  Every C++ program is a word over the alphabet of ascii symbols.
- For a C++ program P, we denote the corresponding word by $\langle P \rangle$.
- Let L_{C++} denote the language of all valid C++ programs.
  (Note: valid = syntactically correct)

C++ programs (2)

- Every C++ program is a piece of text (sequence of ascii symbols).
  Every C++ program is a word over the alphabet of ascii symbols.
- For a C++ program P, we denote the corresponding word by $\langle P \rangle$.
- Let L_{c++} denote the language of all valid C++ programs.
  (Note: valid = syntactically correct)

Fact

The language L_{c++} is Turing-decidable.

Proof: Use a C++ compiler

C++ programs (3)

Fact

There exists a C++ program P^* that behaves as follows.

- P^* takes as input an arbitrary C++ program P and an arbitrary input word w for P
- If P terminates on input w and generates an output w', then also P^* terminates with output w'
- If P does not terminate on input w, then also P^* does not terminate.

Proof: Use an emulator (written in C++) for C++ programs

Remark: In the language of Turing machines, this program P^* is usually called a "**universal Turing machine**"

Undecidability

- Undecidable problems
- The diagonal language
- The complement of the diagonal language
- The sub-program technique
- The halting problem
- The Epsilon-halting problem

The number of computational problems

Every computational problem with inputs encoded in binary corresponds to a language over the alphabet $\{0, 1\}$ (and also the other way round).

The number of computational problems

Every computational problem with inputs encoded in binary corresponds to a language over the alphabet $\{0,1\}$ (and also the other way round).

- Let $\mathcal{L}$ denote the set of all computational problems over $\{0,1\}$.
- A computational problem $L \in \mathcal{L}$ is a subset of $\{0,1\}^*$.
- Hence $\mathcal{L}$ is the set of all subsets of $\{0,1\}^*$.
  Hence $\mathcal{L}$ is the power set of $\{0,1\}^*$.
  Hence $\mathcal{L} = \mathcal{P}(\{0,1\}^*)$.

The number of computational problems

Every computational problem with inputs encoded in binary corresponds to a language over the alphabet $\{0,1\}$ (and also the other way round).

- Let $\mathcal{L}$ denote the set of all computational problems over $\{0,1\}$.
- A computational problem $L \in \mathcal{L}$ is a subset of $\{0,1\}^*$.
- Hence $\mathcal{L}$ is the set of all subsets of $\{0,1\}^*$.
  Hence $\mathcal{L}$ is the power set of $\{0,1\}^*$.
  Hence $\mathcal{L} = \mathcal{P}(\{0,1\}^*)$.

We observe:

- $\{0,1\}^*$ has the same cardinality as $\mathbb{N}$.
- Hence $\mathcal{L} = \mathcal{P}(\{0,1\}^*)$ has the same cardinality as $\mathcal{P}(\mathbb{N})$.
- Therefore the set $\mathcal{L}$ of computational problems is uncountable.

The number of C++ programs

Recall from preceding lectures:

- Every C++ program is a piece of text (sequence of ascii symbols).
  Every C++ program is a word over the alphabet of ascii symbols.

- The set of words $\langle P \rangle$ that describe valid C++ programs is countable.
- The set of valid C++ programs is countable.

The existence of undecidable problems

Let us summarize:

- There exist **uncountably** many languages over $\{0, 1\}$.
- There exist **countably** many C++ programs.

The existence of undecidable problems

Let us summarize:

- There exist **uncountably** many languages over $\{0, 1\}$.
- There exist **countably** many C++ programs.

Immediate consequence

There exist languages that are undecidable.

The existence of undecidable problems

Let us summarize:

- There exist **uncountably** many languages over $\{0, 1\}$.
- There exist **countably** many C++ programs.

Immediate consequence

There exist languages that are undecidable.

At first sight, the sheer existence of undecidable problems does not look very threathening to us:

> We know that these problems do exist, but perhaps they are extremely artificial and contrived, and perhaps they all are far far far away from real life and from real world applications.

The existence of undecidable problems

Let us summarize:

- There exist **uncountably** many languages over $\{0, 1\}$.
- There exist **countably** many C++ programs.

Immediate consequence

There exist languages that are undecidable.

At first sight, the sheer existence of undecidable problems does not look very threathening to us:

> We know that these problems do exist, but perhaps they are extremely artificial and contrived, and perhaps they all are far far far away from real life and from real world applications.

Unfortunately, this is not true:

> We will see that lots of important problems are undecidable.

The diagonal language (1): Definition

Recall our notation on words w_i and programs P_i:

Notation

- The i-th word in the canonical ordering of $\{0,1\}^*$ is denoted w_i.
- The i-th C++ program in the canonical ordering is denoted P_i.

The diagonal language (1): Definition

Recall our notation on words w_i and programs P_i:

Notation

- The i-th word in the canonical ordering of $\{0,1\}^*$ is denoted w_i.
- The i-th C++ program in the canonical ordering is denoted P_i.

Definition

The diagonal language D is given by

$$D = \{ w \in \{0,1\}^* \mid w = w_i, \text{ and } P_i \text{ does not accept } w \}$$

In other words:

The i-th word w_i in the canonical ordering lies in D
if and only if
the i-th C++ program P_i does not accept w_i.

The diagonal language (2): Intuition

Where does the name "**diagonal language**" come from?
Consider an infinite matrix A with

$$A_{i,j} = \begin{cases} 1 & \text{if } P_i \text{ accepts the word } w_j \\ 0 & \text{otherwise} \end{cases}$$

The diagonal language (2): Intuition

Where does the name "**diagonal language**" come from?
Consider an infinite matrix A with

$$A_{i,j} = \begin{cases} 1 & \text{if } P_i \text{ accepts the word } w_j \\ 0 & \text{otherwise} \end{cases}$$

Example

	w_0	w_1	w_2	w_3	w_4	
P_0	**0**	1	1	0	1	$\cdots$
P_1	1	**0**	1	0	1	$\cdots$
P_2	0	0	**1**	0	1	$\cdots$
P_3	0	1	1	**1**	0	$\cdots$
P_4	0	1	0	0	**0**	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$		

The diagonal language follows the diagonal of matrix A:

$$D = \{w_i \mid A_{i,i} = 0\}$$

The diagonal language (3): The proof

Theorem

The diagonal language D is undecidable.

The diagonal language (3): The proof

Theorem

The diagonal language D is undecidable.

Proof:

- Let us assume for the sake of contradiction that D is decidable. Then there exists a C++ program P_i that decides D.

The diagonal language (3): The proof

Theorem

The diagonal language D is undecidable.

Proof:

- Let us assume for the sake of contradiction that D is decidable.
  Then there exists a C++ program P_i that decides D.
- We feed input w_i into program P_i, and we distinguish two cases on the resulting outcome.

The diagonal language (3): The proof

Theorem

The diagonal language D is undecidable.

Proof:

- Let us assume for the sake of contradiction that D is decidable.
  Then there exists a C++ program P_i that decides D.
- We feed input w_i into program P_i, and we distinguish two cases on the resulting outcome.
- **Case 1:** If $w_i \in D$, then $w_i \in L(P_i)$ and hence P_i accepts w_i.
  But then by definition of D, we have $w_i \notin D$. **Contradiction!**

The diagonal language (3): The proof

Theorem

The diagonal language D is undecidable.

Proof:

- Let us assume for the sake of contradiction that D is decidable.
  Then there exists a C++ program P_i that decides D.
- We feed input w_i into program P_i, and we distinguish two cases on the resulting outcome.
- **Case 1:** If $w_i \in D$, then $w_i \in L(P_i)$ and hence P_i accepts w_i.
  But then by definition of D, we have $w_i \notin D$. **Contradiction!**
- **Case 2:** If $w_i \notin D$, then $w_i \notin L(P_i)$ and P_i does not accept w_i.
  But then by definition of D, we have $w_i \in D$. **Contradiction!**

The diagonal language (3): The proof

Theorem

The diagonal language D is undecidable.

Proof:

- Let us assume for the sake of contradiction that D is decidable.
  Then there exists a C++ program P_i that decides D.
- We feed input w_i into program P_i, and we distinguish two cases on the resulting outcome.
- **Case 1:** If $w_i \in D$, then $w_i \in L(P_i)$ and hence P_i accepts w_i.
  But then by definition of D, we have $w_i \notin D$. **Contradiction!**
- **Case 2:** If $w_i \notin D$, then $w_i \notin L(P_i)$ and P_i does not accept w_i.
  But then by definition of D, we have $w_i \in D$. **Contradiction!**
- We conclude that D is undecidable. $\square$

Complement of diagonal language (1)

Definition

The complement $\overline{D}$ of the diagonal language is given by

$$\overline{D} = \{ w \in \{0,1\}^* \mid w = w_i \text{ and } P_i \text{ does accept } w \}$$

Complement of diagonal language (1)

Definition

The complement $\overline{D}$ of the diagonal language is given by

$$\overline{D} = \{ w \in \{0,1\}^* \mid w = w_i \text{ and } P_i \text{ does accept } w \}$$

Theorem

The complement $\overline{D}$ of the diagonal language is undecidable.

Complement of diagonal language (1)

Definition

The complement $\overline{D}$ of the diagonal language is given by

$$\overline{D} = \{ w \in \{0,1\}^* \mid w = w_i \text{ and } P_i \text{ does accept } w \}$$

Theorem

The complement $\overline{D}$ of the diagonal language is undecidable.

Proof:

- Let us assume for the sake of contradiction that $\overline{D}$ is decidable.

Complement of diagonal language (1)

Definition

The complement $\overline{D}$ of the diagonal language is given by

$$\overline{D} = \{ w \in \{0,1\}^* \mid w = w_i \text{ and } P_i \text{ does accept } w \}$$

Theorem

The complement $\overline{D}$ of the diagonal language is undecidable.

Proof:

- Let us assume for the sake of contradiction that $\overline{D}$ is decidable.
- Then there exists a C++ program P that decides $\overline{D}$.
  P halts on every input w, and accepts w if and only if $w \in \overline{D}$.

Complement of diagonal language (1)

Definition

The complement $\overline{D}$ of the diagonal language is given by

$$\overline{D} = \{ w \in \{0,1\}^* \mid w = w_i \text{ and } P_i \text{ does accept } w \}$$

Theorem

The complement $\overline{D}$ of the diagonal language is undecidable.

Proof:

- Let us assume for the sake of contradiction that $\overline{D}$ is decidable.
- Then there exists a C++ program P that decides $\overline{D}$.
  P halts on every input w, and accepts w if and only if $w \in \overline{D}$.
- We construct a new C++ program P' that uses program P as a sub-program: P' first runs P on the input, and then negates the resulting output of P.

Complement of diagonal language (1)

Definition

The complement $\overline{D}$ of the diagonal language is given by

$$\overline{D} = \{ w \in \{0,1\}^* \mid w = w_i \text{ and } P_i \text{ does accept } w \}$$

Theorem

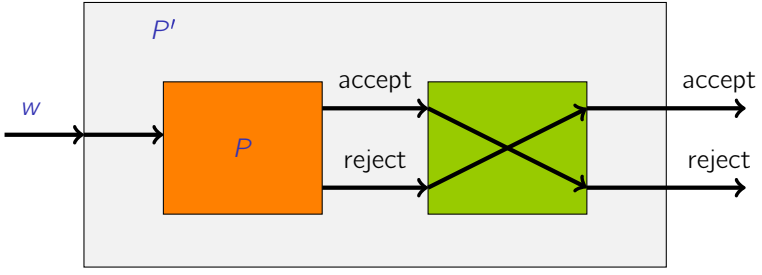
The complement $\overline{D}$ of the diagonal language is undecidable.

Proof:

- Let us assume for the sake of contradiction that $\overline{D}$ is decidable.
- Then there exists a C++ program P that decides $\overline{D}$.
  P halts on every input w, and accepts w if and only if $w \in \overline{D}$.
- We construct a new C++ program P' that uses program P as a sub-program: P' first runs P on the input, and then negates the resulting output of P.
- The new C++ program P' decides D. **Contradiction!**

Complement of diagonal language (2)

Illustration: How to construct program P' from program P.



The existence of P collides with the undecidability of D.
Since program P' does not exist, program P cannot exist.
Hence $\overline{D}$ is undecidable.

Sub-program technique (1)

The proof technique from the preceding theorem (that gave us the undecidability of $\overline{D}$) can be summarized as follows:

Sub-program technique (1)

The proof technique from the preceding theorem (that gave us the undecidability of $\overline{D}$) can be summarized as follows:

Sub-program technique for proving undecidability

- Let L' be a known undecidable language.
- Let L be a new language under current investigation.

In order to establish the undecidability of L, it is sufficient to show that by using a sub-program P for deciding language L we can build a C++ program for deciding language L'.

Sub-program technique (1)

The proof technique from the preceding theorem (that gave us the undecidability of $\overline{D}$) can be summarized as follows:

Sub-program technique for proving undecidability

- Let L' be a known undecidable language.
- Let L be a new language under current investigation.

In order to establish the undecidability of L, it is sufficient to show
that by using a sub-program P for deciding language L
we can build a C++ program for deciding language L'.

In the following slides, we provide illustrations for the sub-program technique and apply it to many example languages.

Sub-program technique (2)

Observation

If language $L \subseteq \{0,1\}^*$ is **undecidable**,
then also its complement $\overline{L}$ is **undecidable**.

Sub-program technique (2)

Observation

If language $L \subseteq \{0,1\}^*$ is **undecidable**,
then also its complement $\overline{L}$ is **undecidable**.

Observation

If language $L \subseteq \{0,1\}^*$ is **decidable**,
then also its complement $\overline{L}$ is **decidable**.

The halting problem (1)

The **halting problem** asks us to decide,
whether a given C++ program halts on a given input.

Definition

The halting problem H is given by $H = \{\langle P \rangle w \mid P \text{ halts on } w\}$.

The halting problem (1)

The **halting problem** asks us to decide,
whether a given C++ program halts on a given input.

Definition

The halting problem H is given by $H = \{\langle P \rangle w \mid P \text{ halts on } w\}$.

It would be great, if a C++ compiler could decide the halting problem.
However: We will see that H is undecidable.

The halting problem (2)

Theorem

The halting problem H is undecidable.

Proof idea (The details are on the following slides):
We apply the sub-program technique.

The halting problem (2)

Theorem

The halting problem H is undecidable.

Proof idea (The details are on the following slides):
We apply the sub-program technique.

- Suppose for the sake of contradiction that there exists a C++ program P_H that decides H: P_H halts on each input, and it only accepts inputs of the form $\langle P \rangle w$, for which P halts on w.

The halting problem (2)

Theorem

The halting problem H is undecidable.

Proof idea (The details are on the following slides):
We apply the sub-program technique.

- Suppose for the sake of contradiction that there exists a C++ program P_H that decides H: P_H halts on each input, and it only accepts inputs of the form $\langle P \rangle w$, for which P halts on w.
- We construct a new C++ program $P_{\overline{D}}$ that uses P_H as a sub-program and that decides $\overline{D}$.

The halting problem (2)

Theorem

The halting problem H is undecidable.

Proof idea (The details are on the following slides):
We apply the sub-program technique.

- Suppose for the sake of contradiction that there exists a C++ program P_H that decides H: P_H halts on each input, and it only accepts inputs of the form $\langle P \rangle w$, for which P halts on w.
- We construct a new C++ program $P_{\overline{D}}$ that uses P_H as a sub-program and that decides $\overline{D}$.
- This contradicts the undecidability of $\overline{D}$,
  and hence implies the non-existence of the C++ program P_H.

The full proof (1a)

The C++ program $P_{\overline{D}}$ with sub-program P_H:

(1) For input w, first determine the index i with $w = w_i$

The full proof (1a)

The C++ program $P_{\overline{D}}$ with sub-program P_H:

(1) For input w, first determine the index i with $w = w_i$

(2) Next determine the i-th program $\langle P_i \rangle$ in canonical ordering

The full proof (1a)

The C++ program $P_{\overline{D}}$ with sub-program P_H:

(1) For input w, first determine the index i with $w = w_i$

(2) Next determine the i-th program $\langle P_i \rangle$ in canonical ordering

(3) Call sub-program P_H on the input $\langle P_i \rangle w_i$.

The full proof (1a)

The C++ program $P_{\overline{D}}$ with sub-program P_H:

(1) For input w, first determine the index i with $w = w_i$

(2) Next determine the i-th program $\langle P_i \rangle$ in canonical ordering

(3) Call sub-program P_H on the input $\langle P_i \rangle w_i$.

(3a) If P_H rejects, then reject the input.

The full proof (1a)

The C++ program $P_{\overline{D}}$ with sub-program P_H:

(1) For input w, first determine the index i with $w = w_i$

(2) Next determine the i-th program $\langle P_i \rangle$ in canonical ordering

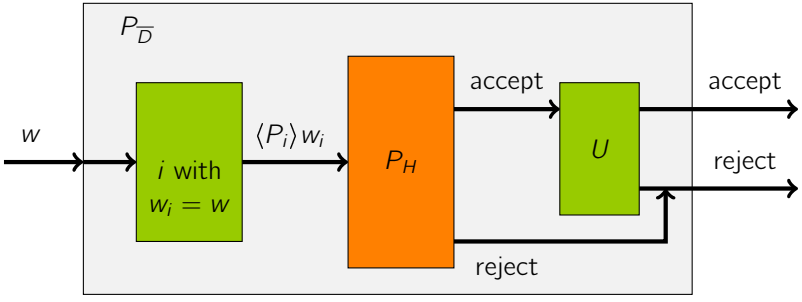
(3) Call sub-program P_H on the input $\langle P_i \rangle w_i$.

(3a) If P_H rejects, then reject the input.

(3b) If P_H accepts, then simulate the behavior of program P_i on input w_i (with a C++ emulator written in C++)

The full proof (1b)

Illustration: How to construct program $P_{\overline{D}}$ from program P_H.



The existence of $P_{\overline{D}}$ collides with the undecidability of D.
Hence program P_H does not exist.
Hence the halting problem H is undecidable.

The full proof (2)

It remains to show that our new program $P_{\overline{D}}$ indeed works correctly.

For the correctness we need to show:

A. $w \in \overline{D} \Rightarrow P_{\overline{D}}$ accepts w

B. $w \notin \overline{D} \Rightarrow P_{\overline{D}}$ rejects w

The full proof (3)

Let $w = w_i$.

The full proof (3)

Let $w = w_i$.

$w \in \overline{D}$

The full proof (3)

Let $w = w_i$.

$$w \in \overline{D} \quad \Rightarrow \quad P_i \text{ accepts } w_i$$

The full proof (3)

Let $w = w_i$.

$$\begin{aligned} w \in \overline{D} &\Rightarrow P_i \text{ accepts } w_i \\ &\Rightarrow P_H \text{ and } U \text{ accept } \langle P_i \rangle w_i \end{aligned}$$

The full proof (3)

Let $w = w_i$.

$$\begin{aligned} w \in \overline{D} &\Rightarrow P_i \text{ accepts } w_i \\ &\Rightarrow P_H \text{ and } U \text{ accept } \langle P_i \rangle w_i \\ &\Rightarrow P_{\overline{D}} \text{ accepts } w \end{aligned}$$

The full proof (3)

Let $w = w_i$.

$$
\begin{aligned}
w \in \overline{D} &\Rightarrow P_i \text{ accepts } w_i \\
&\Rightarrow P_H \text{ and } U \text{ accept } \langle P_i \rangle w_i \\
&\Rightarrow P_{\overline{D}} \text{ accepts } w \\
&\Rightarrow \text{Part A is done}
\end{aligned}
$$

The full proof (3)

Let $w = w_i$.

$$
\begin{aligned}
w \in \overline{D} \quad &\Rightarrow \quad P_i \text{ accepts } w_i \\
&\Rightarrow \quad P_H \text{ and } U \text{ accept } \langle P_i \rangle w_i \\
&\Rightarrow \quad P_{\overline{D}} \text{ accepts } w \\
&\Rightarrow \quad \text{Part A is done}
\end{aligned}
$$

$$
w \notin \overline{D} \quad \Rightarrow \quad P_i \text{ does not accept } w_i
$$

The full proof (3)

Let $w = w_i$.

$$
\begin{aligned}
w \in \overline{D} \quad &\Rightarrow \quad P_i \text{ accepts } w_i \\
&\Rightarrow \quad P_H \text{ and } U \text{ accept } \langle P_i \rangle w_i \\
&\Rightarrow \quad P_{\overline{D}} \text{ accepts } w \\
&\Rightarrow \quad \text{Part A is done}
\end{aligned}
$$

$$
\begin{aligned}
w \notin \overline{D} \quad &\Rightarrow \quad P_i \text{ does not accept } w_i \\
&\Rightarrow \quad (\, P_i \text{ does not halt on } w_i \,) \text{ or } (\, P_i \text{ rejects } w_i \,)
\end{aligned}
$$

The full proof (3)

Let $w = w_i$.

$$
\begin{aligned}
w \in \overline{D} \quad &\Rightarrow \quad P_i \text{ accepts } w_i \\
&\Rightarrow \quad P_H \text{ and } U \text{ accept } \langle P_i \rangle w_i \\
&\Rightarrow \quad P_{\overline{D}} \text{ accepts } w \\
&\Rightarrow \quad \text{Part A is done}
\end{aligned}
$$

$$
\begin{aligned}
w \notin \overline{D} \quad &\Rightarrow \quad P_i \text{ does not accept } w_i \\
&\Rightarrow \quad (\, P_i \text{ does not halt on } w_i \,) \text{ or } (\, P_i \text{ rejects } w_i \,) \\
&\Rightarrow \quad (\, P_H \text{ rejects } \langle P_i \rangle w_i \,) \text{ or } (\, P_H \text{ accepts and } U \text{ rejects } \langle P_i \rangle w_i \,)
\end{aligned}
$$

The full proof (3)

Let $w = w_i$.

$$
\begin{aligned}
w \in \overline{D} \quad &\Rightarrow \quad P_i \text{ accepts } w_i \\
&\Rightarrow \quad P_H \text{ and } U \text{ accept } \langle P_i \rangle w_i \\
&\Rightarrow \quad P_{\overline{D}} \text{ accepts } w \\
&\Rightarrow \quad \text{Part A is done}
\end{aligned}
$$

$$
\begin{aligned}
w \notin \overline{D} \quad &\Rightarrow \quad P_i \text{ does not accept } w_i \\
&\Rightarrow \quad (\, P_i \text{ does not halt on } w_i \,) \text{ or } (\, P_i \text{ rejects } w_i \,) \\
&\Rightarrow \quad (\, P_H \text{ rejects } \langle P_i \rangle w_i \,) \text{ or } (\, P_H \text{ accepts and } U \text{ rejects } \langle P_i \rangle w_i \,) \\
&\Rightarrow \quad P_{\overline{D}} \text{ rejects } w \\
&\Rightarrow \quad \text{Part B is done}
\end{aligned}
$$

The Epsilon-halting problem

Definition

The Epsilon-halting problem H_ϵ is given by

$$H_\epsilon = \{\langle P \rangle \mid P \text{ halts on the empty input } \epsilon \}$$

The Epsilon-halting problem

Definition

The Epsilon-halting problem H_ϵ is given by

$$H_\epsilon = \{\langle P \rangle \mid P \text{ halts on the empty input } \epsilon \}$$

Theorem

The Epsilon-halting problem H_ϵ is undecidable.

The Epsilon-halting problem

Definition

The Epsilon-halting problem H_ϵ is given by

$$H_\epsilon = \{\langle P \rangle \mid P \text{ halts on the empty input } \epsilon \}$$

Theorem

The Epsilon-halting problem H_ϵ is undecidable.

Proof idea: We apply the sub-program technique.
With the help of a C++ sub-program P_ϵ that decides H_ϵ, we construct a new C++ program P_H, that decides the undecidable halting problem H.

The proof (1a)

The new C++ program P_H with sub-program P_ϵ works as follows:

(1) If the input is not of the form $\langle P \rangle w$, then P_H rejects.

The proof (1a)

The new C++ program P_H with sub-program P_ϵ works as follows:

(1) If the input is not of the form $\langle P \rangle w$, then P_H rejects.

(2) Otherwise the input is of the form $\langle P \rangle w$, and we formulate a new C++ program P_w^* with the following properties:

The proof (1a)

The new C++ program P_H with sub-program P_ϵ works as follows:

(1) If the input is not of the form $\langle P \rangle w$, then P_H rejects.

(2) Otherwise the input is of the form $\langle P \rangle w$, and we formulate a new C++ program P_w^* with the following properties:

- If P_w^* receives the input ϵ then it simulates the behavior of C++ program P on input w.

The proof (1a)

The new C++ program P_H with sub-program P_ϵ works as follows:

(1) If the input is not of the form $\langle P \rangle w$, then P_H rejects.

(2) Otherwise the input is of the form $\langle P \rangle w$, and we formulate a new C++ program P_w^* with the following properties:

- If P_w^* receives the input ϵ then it simulates the behavior of C++ program P on input w.
- For inputs that are not ϵ, the behavior of P_w^* is irrelevant; for instance, we may assume that it halts right away.

The proof (1a)

The new C++ program P_H with sub-program P_ϵ works as follows:

(1) If the input is not of the form $\langle P \rangle w$, then P_H rejects.

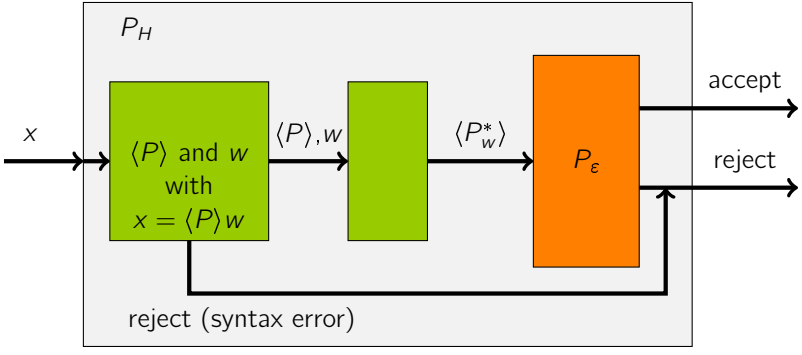
(2) Otherwise the input is of the form $\langle P \rangle w$, and we formulate a new C++ program P_w^* with the following properties:

- If P_w^* receives the input ϵ then it simulates the behavior of C++ program P on input w.
- For inputs that are not ϵ, the behavior of P_w^* is irrelevant; for instance, we may assume that it halts right away.

(3) Now P_H simulates program P_ϵ on the input $\langle P_w^* \rangle$.
P_H accepts/rejects exactly if P_ϵ accepts/rejects.

The proof (1b)

Illustration: How to construct program P_H from program P_ϵ.



The existence of P_H collides with the undecidability of P_H.
Hence program P_ϵ does not exist.
Hence the Epsilon-halting problem H_ϵ is undecidable.

The proof (2)

We show next that our new program P_H indeed works correctly.

- If the input x is not of the form $x = \langle P \rangle w$, then P_H rejects the input (and behaves correctly)
- Hence we assume that $x = \langle P \rangle w$

For the correctness we need to show:

A. $\langle P \rangle w \in H \Rightarrow P_H$ accepts $\langle P \rangle w$

B. $\langle P \rangle w \notin H \Rightarrow P_H$ rejects $\langle P \rangle w$

The proof (3)

$\langle P \rangle w \in H \quad \Rightarrow \quad P$ halts on input w

The proof (3)

$$\begin{aligned} \langle P \rangle w \in H &\Rightarrow P \text{ halts on input } w \\ &\Rightarrow P^*_w \text{ halts on input } \epsilon \end{aligned}$$

The proof (3)

$$\begin{aligned} \langle P \rangle w \in H &\Rightarrow P \text{ halts on input } w \\ &\Rightarrow P^*_w \text{ halts on input } \epsilon \\ &\Rightarrow P_\epsilon \text{ accepts } \langle P^*_w \rangle \end{aligned}$$

The proof (3)

$$
\begin{aligned}
\langle P \rangle w \in H \quad &\Rightarrow \quad P \text{ halts on input } w \\
&\Rightarrow \quad P^*_w \text{ halts on input } \epsilon \\
&\Rightarrow \quad P_\epsilon \text{ accepts } \langle P^*_w \rangle \\
&\Rightarrow \quad P_H \text{ accepts } \langle P \rangle w
\end{aligned}
$$

The proof (3)

$$
\begin{aligned}
\langle P \rangle w \in H \quad &\Rightarrow \quad P \text{ halts on input } w \\
&\Rightarrow \quad P^*_w \text{ halts on input } \epsilon \\
&\Rightarrow \quad P_\epsilon \text{ accepts } \langle P^*_w \rangle \\
&\Rightarrow \quad P_H \text{ accepts } \langle P \rangle w \\
&\Rightarrow \quad \text{Part A is done}
\end{aligned}
$$

The proof (3)

$$
\begin{aligned}
\langle P \rangle w \in H \quad &\Rightarrow \quad P \text{ halts on input } w \\
&\Rightarrow \quad P^*_w \text{ halts on input } \epsilon \\
&\Rightarrow \quad P_\epsilon \text{ accepts } \langle P^*_w \rangle \\
&\Rightarrow \quad P_H \text{ accepts } \langle P \rangle w \\
&\Rightarrow \quad \text{Part A is done}
\end{aligned}
$$

$$
\langle P \rangle w \notin H \quad \Rightarrow \quad P \text{ does not halt on input } w
$$

The proof (3)

$$
\begin{aligned}
\langle P \rangle w \in H \quad &\Rightarrow \quad P \text{ halts on input } w \\
&\Rightarrow \quad P^*_w \text{ halts on input } \epsilon \\
&\Rightarrow \quad P_\epsilon \text{ accepts } \langle P^*_w \rangle \\
&\Rightarrow \quad P_H \text{ accepts } \langle P \rangle w \\
&\Rightarrow \quad \text{Part A is done}
\end{aligned}
$$

$$
\begin{aligned}
\langle P \rangle w \notin H \quad &\Rightarrow \quad P \text{ does not halt on input } w \\
&\Rightarrow \quad P^*_w \text{ does not halt on input } \epsilon
\end{aligned}
$$

The proof (3)

$$
\begin{aligned}
\langle P \rangle w \in H &\Rightarrow P \text{ halts on input } w \\
&\Rightarrow P^*_w \text{ halts on input } \epsilon \\
&\Rightarrow P_\epsilon \text{ accepts } \langle P^*_w \rangle \\
&\Rightarrow P_H \text{ accepts } \langle P \rangle w \\
&\Rightarrow \text{Part A is done}
\end{aligned}
$$

$$
\begin{aligned}
\langle P \rangle w \notin H &\Rightarrow P \text{ does not halt on input } w \\
&\Rightarrow P^*_w \text{ does not halt on input } \epsilon \\
&\Rightarrow P_\epsilon \text{ rejects } \langle P^*_w \rangle
\end{aligned}
$$

The proof (3)

$$
\begin{aligned}
\langle P \rangle w \in H \quad &\Rightarrow \quad P \text{ halts on input } w \\
&\Rightarrow \quad P^*_w \text{ halts on input } \epsilon \\
&\Rightarrow \quad P_\epsilon \text{ accepts } \langle P^*_w \rangle \\
&\Rightarrow \quad P_H \text{ accepts } \langle P \rangle w \\
&\Rightarrow \quad \text{Part A is done}
\end{aligned}
$$

$$
\begin{aligned}
\langle P \rangle w \notin H \quad &\Rightarrow \quad P \text{ does not halt on input } w \\
&\Rightarrow \quad P^*_w \text{ does not halt on input } \epsilon \\
&\Rightarrow \quad P_\epsilon \text{ rejects } \langle P^*_w \rangle \\
&\Rightarrow \quad P_H \text{ rejects } \langle P \rangle w
\end{aligned}
$$

The proof (3)

$$
\begin{aligned}
\langle P \rangle w \in H \quad &\Rightarrow \quad P \text{ halts on input } w \\
&\Rightarrow \quad P^*_w \text{ halts on input } \epsilon \\
&\Rightarrow \quad P_\epsilon \text{ accepts } \langle P^*_w \rangle \\
&\Rightarrow \quad P_H \text{ accepts } \langle P \rangle w \\
&\Rightarrow \quad \text{Part A is done}
\end{aligned}
$$

$$
\begin{aligned}
\langle P \rangle w \notin H \quad &\Rightarrow \quad P \text{ does not halt on input } w \\
&\Rightarrow \quad P^*_w \text{ does not halt on input } \epsilon \\
&\Rightarrow \quad P_\epsilon \text{ rejects } \langle P^*_w \rangle \\
&\Rightarrow \quad P_H \text{ rejects } \langle P \rangle w \\
&\Rightarrow \quad \text{Part B is done}
\end{aligned}
$$

Undecidable versus decidable (1)

We have seen that the following problems are undecidable:

- Given $\langle P \rangle$ and w, does P halt on input w?
- Given $\langle P \rangle$, does P halt on input ϵ?

Undecidable versus decidable (1)

We have seen that the following problems are undecidable:

- Given $\langle P \rangle$ and w, does P halt on input w?
- Given $\langle P \rangle$, does P halt on input ϵ?

Analogous arguments show that the following problems are undecidable:

Exercise

- Given $\langle P \rangle$ and w, does $w \in L(P)$ hold?
- Given $\langle P \rangle$, does $\varepsilon \in L(P)$ hold?
- Given $\langle P \rangle$, does $\langle P \rangle \in L(P)$ hold?
- Given $\langle P \rangle$, is $L(P)$ empty?
- Given $\langle P \rangle$, is $L(P) = \Sigma^*$?
- Given $\langle P \rangle$, is $L(P)$ finite?
- Given $\langle P \rangle$, is $L(P)$ regular?
- Given $\langle P \rangle$, is $L(P)$ context-free?

Undecidable versus decidable (2)

On the other hand, the following problems are decidable:

- Given $\langle P \rangle$, is $L(P) \subseteq \Sigma^*$?
- Given $\langle P \rangle$, is $L(P)$ recognized by some C++ program?
- Given $\langle P \rangle$, does P use an even number of variables?
- Given $\langle P \rangle$, does $\langle P \rangle$ have even length?
- Given $\langle P \rangle$, is P a Java program?

Partial functions

In general, C++ programs will not halt for every input. They compute so-called **partial functions**, which are formalized as follows:

- A C++ program P computes a function of the form

$$f_P \colon \{0,1\}^* \to \{0,1\}^* \cup \{\bot\}$$

The symbol $\bot$ stands for "**undefined**" which means that the program does not halt.

Partial functions

In general, C++ programs will not halt for every input. They compute so-called **partial functions**, which are formalized as follows:

- A C++ program P computes a function of the form

  $$f_P \colon \{0,1\}^* \to \{0,1\}^* \cup \{\bot\}$$

  The symbol $\bot$ stands for "**undefined**" which means that the program does not halt.
- In the case of decision problems the function is of the form

  $$f_P \colon \{0,1\}^* \to \{0,1,\bot\}$$

  Here 0 stands for "**reject**, while 1 stands for "**accept**", and $\bot$ stands for "**does-not-halt**".

Theorem of Rice

Theorem

Let $\mathcal{R}$ be the set of partial functions computable by C++ programs.
Let $\mathcal{S}$ be some subset of $\mathcal{R}$ with $\emptyset \subsetneq \mathcal{S} \subsetneq \mathcal{R}$.

Then the language

$$L(\mathcal{S}) = \{\langle P \rangle \mid P \text{ computes a function in } \mathcal{S}\}$$

is undecidable.

Theorem of Rice

Theorem

Let $\mathcal{R}$ be the set of partial functions computable by C++ programs.
Let $\mathcal{S}$ be some subset of $\mathcal{R}$ with $\emptyset \subsetneq \mathcal{S} \subsetneq \mathcal{R}$.

Then the language

$$L(\mathcal{S}) = \{\langle P \rangle \mid P \text{ computes a function in } \mathcal{S}\}$$

is undecidable.

In other words: All non-trivial statements on functions computed by C++ programs are undecidable.

Application (1)

Example 1

- Let $\mathcal{S} = \{f_P \mid f_P(\epsilon) \neq \perp\}$.

Application (1)

Example 1

- Let $\mathcal{S} = \{f_P \mid f_P(\epsilon) \neq \perp\}$.
- Then

$$\begin{aligned} L(\mathcal{S}) &= \{\langle P \rangle \mid P \text{ computes a function in } \mathcal{S}\} \\ &= \{\langle P \rangle \mid P \text{ halts on input } \epsilon\} \\ &= H_\epsilon \end{aligned}$$

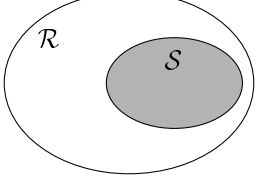
Application (1)

Example 1

- Let $\mathcal{S} = \{f_P \mid f_P(\epsilon) \neq \perp\}$.
- Then

$$\begin{aligned} L(\mathcal{S}) &= \{\langle P \rangle \mid P \text{ computes a function in } \mathcal{S}\} \\ &= \{\langle P \rangle \mid P \text{ halts on input } \epsilon\} \\ &= H_\epsilon \end{aligned}$$

- The theorem of Rice yields that the Epsilon-halting problem H_ϵ is undecidable. (But we already knew about that...)



Application II

Example 2

- Let $\mathcal{S} = \{f_P \mid \forall w \in \{0,1\}^* \colon f_P(w) \neq \perp\}$.
- Then

$$\begin{aligned} L(\mathcal{S}) &= \{\langle P \rangle \mid P \text{ computes a function in } \mathcal{S}\} \\ &= \{\langle P \rangle \mid P \text{ halts on every input}\} \end{aligned}$$

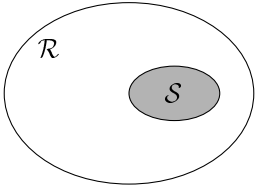
Application II

Example 2

- Let $\mathcal{S} = \{f_P \mid \forall w \in \{0,1\}^* \colon f_P(w) \neq \perp\}$.
- Then

$$\begin{aligned} L(\mathcal{S}) &= \{\langle P \rangle \mid P \text{ computes a function in } \mathcal{S}\} \\ &= \{\langle P \rangle \mid P \text{ halts on every input}\} \end{aligned}$$

- This language is also known as the **total halting problem** H_{tot}.
- The theorem of Rice yields that H_{tot} is undecidable.



Application (3)

Example 3

- Let $\mathcal{S} = \{f_P \mid \forall w \in \{0,1\}^* \colon f_P(w) = 1\}$.

Application (3)

Example 3

- Let $\mathcal{S} = \{f_P \mid \forall w \in \{0,1\}^* \colon f_P(w) = 1\}$.
- Then

$$\begin{aligned} L(\mathcal{S}) &= \{\langle P \rangle \mid P \text{ computes a function in } \mathcal{S}\} \\ &= \{\langle P \rangle \mid P \text{ halts on every input with output } 1\} \end{aligned}$$

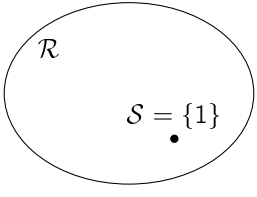
Application (3)

Example 3

- Let $\mathcal{S} = \{f_P \mid \forall w \in \{0,1\}^* \colon f_P(w) = 1\}$.
- Then

$$\begin{aligned} L(\mathcal{S}) &= \{\langle P \rangle \mid P \text{ computes a function in } \mathcal{S}\} \\ &= \{\langle P \rangle \mid P \text{ halts on every input with output } 1\} \end{aligned}$$

- The theorem of Rice yields that $L(\mathcal{S})$ is undecidable.



Proof of Rice (0)

Here is once again the statement of the theorem of Rice:

Theorem

Let $\mathcal{R}$ be the set of partial functions computable by C++ programs.
Let $\mathcal{S}$ be some subset of $\mathcal{R}$ with $\emptyset \subsetneq \mathcal{S} \subsetneq \mathcal{R}$.

Then the language

$$L(\mathcal{S}) = \{\langle P \rangle \mid P \text{ computes a function in } \mathcal{S}\}$$

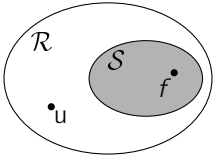
is undecidable.

Proof of Rice (1)

We apply the sub-program technique: With the help of a C++ program $P_{L(\mathcal{S})}$ that decides $L(\mathcal{S})$, we will construct a new C++ program P_ϵ that decides the (undecidable) Epsilon-halting problem H_ϵ.

Let us first agree on the following:

- Let u be the nowhere defined function $u(w) \equiv \perp$.
- Without loss of generality we assume $u \notin \mathcal{S}$.

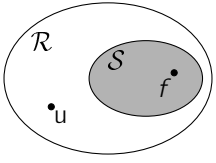


Proof of Rice (1)

We apply the sub-program technique: With the help of a C++ program $P_{L(\mathcal{S})}$ that decides $L(\mathcal{S})$, we will construct a new C++ program P_ϵ that decides the (undecidable) Epsilon-halting problem H_ϵ.

Let us first agree on the following:

- Let u be the nowhere defined function $u(w) \equiv \perp$.
- Without loss of generality we assume $u \notin \mathcal{S}$.



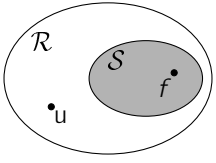
Remark: In the case $u \in \mathcal{S}$, we simply consider $\mathcal{R} \setminus \mathcal{S}$ instead of $\mathcal{S}$, and show the undecidability of $L(\mathcal{R} \setminus \mathcal{S})$. This implies the undecidability of $L(\mathcal{S})$.

Proof of Rice (1)

We apply the sub-program technique: With the help of a C++ program $P_{L(\mathcal{S})}$ that decides $L(\mathcal{S})$, we will construct a new C++ program P_ϵ that decides the (undecidable) Epsilon-halting problem H_ϵ.

Let us first agree on the following:

- Let u be the nowhere defined function $u(w) \equiv \perp$.
- Without loss of generality we assume $u \notin \mathcal{S}$.



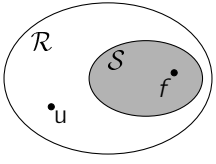
Remark: In the case $u \in \mathcal{S}$, we simply consider $\mathcal{R} \setminus \mathcal{S}$ instead of $\mathcal{S}$, and show the undecidability of $L(\mathcal{R} \setminus \mathcal{S})$. This implies the undecidability of $L(\mathcal{S})$.

Proof of Rice (1)

We apply the sub-program technique: With the help of a C++ program $P_{L(\mathcal{S})}$ that decides $L(\mathcal{S})$, we will construct a new C++ program P_ϵ that decides the (undecidable) Epsilon-halting problem H_ϵ.

Let us first agree on the following:

- Let u be the nowhere defined function $u(w) \equiv \perp$.
- Without loss of generality we assume $u \notin \mathcal{S}$.
- Let f be a function in $\mathcal{S}$.
- Let Q_f be a program that computes f.



Remark: In the case $u \in \mathcal{S}$, we simply consider $\mathcal{R} \setminus \mathcal{S}$ instead of $\mathcal{S}$, and show the undecidability of $L(\mathcal{R} \setminus \mathcal{S})$. This implies the undecidability of $L(\mathcal{S})$.

Proof of Rice (2a)

The new C++ program P_ϵ with sub-program $P_{L(\mathcal{S})}$ works as follows:

(1) If the input is not of the form $\langle P \rangle$,
then P_ϵ rejects the input right away

Proof of Rice (2a)

The new C++ program P_ϵ with sub-program $P_{L(S)}$ works as follows:

(1) If the input is not of the form $\langle P \rangle$,
then P_ϵ rejects the input right away

(2) Otherwise the input is of the form $\langle P \rangle$, and we formulate a new C++ program P^* shown below

Proof of Rice (2a)

The new C++ program P_ϵ with sub-program $P_{L(\mathcal{S})}$ works as follows:

(1) If the input is not of the form $\langle P \rangle$,
then P_ϵ rejects the input right away

(2) Otherwise the input is of the form $\langle P \rangle$, and we formulate a new C++ program P^* shown below

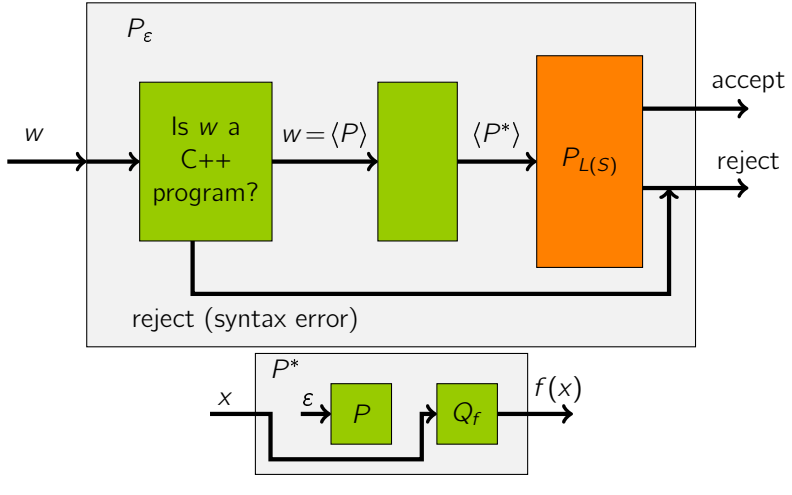
(3) Finally we run $P_{L(\mathcal{S})}$ on the input $\langle P^* \rangle$.
We accept/reject exactly if $P_{L(\mathcal{S})}$ accepts/rejects.

Behavior of program P^* on input x

Phase A: Simulate the behavior of P on input ϵ

Phase B: Simulate the behavior of Q_f on input x.
Halt as soon as Q_f halts, and pass on the output.

Proof of Rice (2b)



Proof of Rice (3)

For an input $w = \langle P \rangle$ we have:

$$w \in H_\epsilon \quad \Rightarrow \quad P \text{ halts on input } \epsilon$$

Proof of Rice (3)

For an input $w = \langle P \rangle$ we have:

$$\begin{aligned} w \in H_\epsilon &\Rightarrow P \text{ halts on input } \epsilon \\ &\Rightarrow P^* \text{ computes } f \end{aligned}$$

Proof of Rice (3)

For an input $w = \langle P \rangle$ we have:

$$\begin{aligned} w \in H_\epsilon &\Rightarrow P \text{ halts on input } \epsilon \\ &\Rightarrow P^* \text{ computes } f \\ &\Rightarrow \langle P^* \rangle \in L(\mathcal{S}) \qquad \text{(as } f \in \mathcal{S} \text{ holds)} \end{aligned}$$

Proof of Rice (3)

For an input $w = \langle P \rangle$ we have:

$$
\begin{aligned}
w \in H_\epsilon \quad &\Rightarrow \quad P \text{ halts on input } \epsilon \\
&\Rightarrow \quad P^* \text{ computes } f \\
&\Rightarrow \quad \langle P^* \rangle \in L(\mathcal{S}) \qquad \text{(as } f \in \mathcal{S} \text{ holds)} \\
&\Rightarrow \quad P_{L(\mathcal{S})} \text{ accepts } \langle P^* \rangle
\end{aligned}
$$

Proof of Rice (3)

For an input $w = \langle P \rangle$ we have:

$$
\begin{aligned}
w \in H_\epsilon &\Rightarrow P \text{ halts on input } \epsilon \\
&\Rightarrow P^* \text{ computes } f \\
&\Rightarrow \langle P^* \rangle \in L(\mathcal{S}) \qquad (\text{as } f \in \mathcal{S} \text{ holds}) \\
&\Rightarrow P_{L(\mathcal{S})} \text{ accepts } \langle P^* \rangle \\
&\Rightarrow P_\epsilon \text{ accepts } w
\end{aligned}
$$

Proof of Rice (3)

For an input $w = \langle P \rangle$ we have:

$$
\begin{aligned}
w \in H_\epsilon \quad &\Rightarrow \quad P \text{ halts on input } \epsilon \\
&\Rightarrow \quad P^* \text{ computes } f \\
&\Rightarrow \quad \langle P^* \rangle \in L(\mathcal{S}) \qquad \text{(as } f \in \mathcal{S} \text{ holds)} \\
&\Rightarrow \quad P_{L(\mathcal{S})} \text{ accepts } \langle P^* \rangle \\
&\Rightarrow \quad P_\epsilon \text{ accepts } w
\end{aligned}
$$

$$
w \notin H_\epsilon \quad \Rightarrow \quad P \text{ does not halt on input } \epsilon
$$

Proof of Rice (3)

For an input $w = \langle P \rangle$ we have:

$$
\begin{aligned}
w \in H_\epsilon &\Rightarrow P \text{ halts on input } \epsilon \\
&\Rightarrow P^* \text{ computes } f \\
&\Rightarrow \langle P^* \rangle \in L(\mathcal{S}) \qquad (\text{as } f \in \mathcal{S} \text{ holds}) \\
&\Rightarrow P_{L(\mathcal{S})} \text{ accepts } \langle P^* \rangle \\
&\Rightarrow P_\epsilon \text{ accepts } w
\end{aligned}
$$

$$
\begin{aligned}
w \notin H_\epsilon &\Rightarrow P \text{ does not halt on input } \epsilon \\
&\Rightarrow P^* \text{ computes } u
\end{aligned}
$$

Proof of Rice (3)

For an input $w = \langle P \rangle$ we have:

$$
\begin{aligned}
w \in H_\epsilon &\Rightarrow P \text{ halts on input } \epsilon \\
&\Rightarrow P^* \text{ computes } f \\
&\Rightarrow \langle P^* \rangle \in L(\mathcal{S}) \qquad (\text{as } f \in \mathcal{S} \text{ holds}) \\
&\Rightarrow P_{L(\mathcal{S})} \text{ accepts } \langle P^* \rangle \\
&\Rightarrow P_\epsilon \text{ accepts } w
\end{aligned}
$$

$$
\begin{aligned}
w \notin H_\epsilon &\Rightarrow P \text{ does not halt on input } \epsilon \\
&\Rightarrow P^* \text{ computes } u \\
&\Rightarrow \langle P^* \rangle \notin L(\mathcal{S}) \qquad (\text{as } u \notin \mathcal{S} \text{ holds})
\end{aligned}
$$

Proof of Rice (3)

For an input $w = \langle P \rangle$ we have:

$$
\begin{aligned}
w \in H_\epsilon &\Rightarrow P \text{ halts on input } \epsilon \\
&\Rightarrow P^* \text{ computes } f \\
&\Rightarrow \langle P^* \rangle \in L(\mathcal{S}) \qquad (\text{as } f \in \mathcal{S} \text{ holds}) \\
&\Rightarrow P_{L(\mathcal{S})} \text{ accepts } \langle P^* \rangle \\
&\Rightarrow P_\epsilon \text{ accepts } w
\end{aligned}
$$

$$
\begin{aligned}
w \notin H_\epsilon &\Rightarrow P \text{ does not halt on input } \epsilon \\
&\Rightarrow P^* \text{ computes } u \\
&\Rightarrow \langle P^* \rangle \notin L(\mathcal{S}) \qquad (\text{as } u \notin \mathcal{S} \text{ holds}) \\
&\Rightarrow P_{L(\mathcal{S})} \text{ rejects } \langle P^* \rangle
\end{aligned}
$$

Proof of Rice (3)

For an input $w = \langle P \rangle$ we have:

$$
\begin{aligned}
w \in H_\epsilon &\Rightarrow P \text{ halts on input } \epsilon \\
&\Rightarrow P^* \text{ computes } f \\
&\Rightarrow \langle P^* \rangle \in L(\mathcal{S}) \qquad (\text{as } f \in \mathcal{S} \text{ holds}) \\
&\Rightarrow P_{L(\mathcal{S})} \text{ accepts } \langle P^* \rangle \\
&\Rightarrow P_\epsilon \text{ accepts } w
\end{aligned}
$$

$$
\begin{aligned}
w \notin H_\epsilon &\Rightarrow P \text{ does not halt on input } \epsilon \\
&\Rightarrow P^* \text{ computes } u \\
&\Rightarrow \langle P^* \rangle \notin L(\mathcal{S}) \qquad (\text{as } u \notin \mathcal{S} \text{ holds}) \\
&\Rightarrow P_{L(\mathcal{S})} \text{ rejects } \langle P^* \rangle \\
&\Rightarrow P_\epsilon \text{ rejects } w
\end{aligned}
$$

Fatal consequences of Rice's theorem

There is no algorithmic method (manual or automated)
that would be able to determine whether a given C++ program
satisfies any (non-trivial) specification.

Analogous statements hold
for all other higher programming languages, as for instance
Java, Pascal, Python, FORTRAN, Algol, LISP, COBOL, etc.

Analogous statements hold
for all programming languages that will be designed in the future

Application (4)

Example 4

- Let $L_{17} = \{\langle P \rangle \mid \text{On input 17, program } P \text{ outputs the number 42}\}$.

Application (4)

Example 4

- Let $L_{17} = \{\langle P \rangle \mid \text{On input 17, program } P \text{ outputs the number 42}\}$.
- Then $L_{17} = L(\mathcal{S})$ for $\mathcal{S} = \{f_P \mid f_P(\mathsf{bin}(17)) = \mathsf{bin}(42)\}$.

Application (4)

Example 4

- Let $L_{17} = \{\langle P \rangle \mid \text{On input 17, program } P \text{ outputs the number 42}\}$.
- Then $L_{17} = L(\mathcal{S})$ for $\mathcal{S} = \{f_P \mid f_P(\mathsf{bin}(17)) = \mathsf{bin}(42)\}$.
- As $\emptyset \subsetneq \mathcal{S} \subsetneq \mathcal{R}$, Rice implies that L_{17} is undecidable.

Application (5)

Example 5

- Let $L_{44} = \{\langle P \rangle \mid$ there exists a word w, so that on input w the program P executes the statement in line 44 at least once $\}$.

Application (5)

Example 5

- Let $L_{44} = \{\langle P \rangle \mid$ there exists a word w, so that on input w the program P executes the statement in line 44 at least once $\}$.
- Rice has no consequences for this language.

Application (5)

Example 5

- Let $L_{44} = \{\langle P \rangle \mid$ there exists a word w, so that on input w the program P executes the statement in line 44 at least once $\}$.
- Rice has no consequences for this language.

Question: Is L_{44} decidable?

Application (6)

Example 6

- Let $L_D = \{\langle P \rangle \mid P \text{ decides the diagonal language}\}$.

Application (6)

Example 6

- Let $L_D = \{\langle P \rangle \mid P \text{ decides the diagonal language}\}$.
- Then $L_D = L(\mathcal{S})$ for $\mathcal{S} = \{f_D\}$ with

$$f_D(w) = \begin{cases} 1 & \text{if } w \in D \\ 0 & \text{otherwise.} \end{cases}$$

Application (6)

Example 6

- Let $L_D = \{\langle P \rangle \mid P \text{ decides the diagonal language}\}$.
- Then $L_D = L(\mathcal{S})$ for $\mathcal{S} = \{f_D\}$ with

$$f_D(w) = \begin{cases} 1 & \text{if } w \in D \\ 0 & \text{otherwise.} \end{cases}$$

- Rice has no consequences for this language!

Application (6)

Example 6

- Let $L_D = \{\langle P \rangle \mid P \text{ decides the diagonal language}\}$.
- Then $L_D = L(\mathcal{S})$ for $\mathcal{S} = \{f_D\}$ with

$$f_D(w) = \begin{cases} 1 & \text{if } w \in D \\ 0 & \text{otherwise.} \end{cases}$$

- Rice has no consequences for this language!
- But: This language is decidable, as $L_D = \{\}$.