Do you remember?

- There are bijections between $\mathbb{N}$ and $\mathbb{Z}$, $(\mathbb{N} \times \mathbb{N})$, and $\text{List}(\mathbb{N})$
- This means we can focus on functions in $\mathbb{N} \rightarrow \mathbb{N}$
- There are countably infinite while programs

Something to think about:

- How many subsets of $\mathbb{N}$ are there?
- What is the relationship between $\mathbb{N} \rightarrow \mathbb{B}$ and $\mathbb{N} \times \mathbb{N}$?
- What is the relationship between $\mathbb{B} \rightarrow \mathbb{N}$ and $\mathbb{N} \times \mathbb{N}$?
- Can we write a while program to compute $\phi_S$ (the coding function for while programs)? What about the rewrite function $\Rightarrow$?
- What is $\gamma^{-1}$? Can we write a while program to compute it?
We have seen the \texttt{while} language as a model of computation.

We have shown that anything we can say about programs that compute functions in $\mathbb{N} \to \mathbb{N}$ can generalise to all \texttt{while} programs and programs we want to write in general. This was \textit{coding}.

We have shown that the set of \texttt{while} programs is \textit{countable}.

Now we explore the limits of \textit{computability}. 
What can a computer do?
What can a computer do?

Run programs
What can’t a computer do? (an informal notion)

- Understand emotions?
- Dream?
- Create beauty?
- Predict the future?
- Feel pain?
- Fall in love?

The question is can we phrase it as a function? and can we write a program?

If not, then a computer cannot solve the problem (This ignores probabilistic or non-deterministic settings)
What can’t a computer do? (an informal notion)

Can a computer:
- Understand emotions?
- Dream?
- Create beauty?
- Predict the future?
- Feel pain?
- Fall in love?

The question is can we phrase it as a function? and can we write a program?

If not, then a computer cannot solve the problem

(This ignores probabilistic or non-deterministic settings)
Definition

A function $f : \mathbb{N} \to \mathbb{N}$ is *computable* if, and only if, there is a *while* program $S$, such that for all states $\sigma$, and $n \in \mathbb{N}$ with $\sigma(x) = n$, then either there is a state $\sigma'$ such that

$$< S, \sigma > \Rightarrow^* \sigma'$$

and

$$\sigma'(x) = f(n)$$

or $f(n)$ is undefined. Where $\Rightarrow^*$ is the *transitive closure* of $\Rightarrow$.

Note that if $f(n)$ is undefined we need $< S, \sigma >$ not to terminate.
The previous formal definition means

A function is computable if you can write a \texttt{while} program to compute it.

i.e. it uses the operational semantics to capture what it means for a \texttt{while} program to compute a function. Next (after Easter) we meet another way of doing this.
Examples of Computable Functions

\[ f(x) = 1 \] is computable using \( x := 1 \)

\[ f(x) = x + 1 \] is computable using \( x := x + 1 \)

\[ f(x) = \begin{cases} 
-\frac{x}{x} & \text{if } x < 0 \\
\frac{x}{x} & \text{otherwise}
\end{cases} \] is computable using \( \text{if } x < 0 \text{ then } x := -x \)

Suppose \( f \) and \( g \) are computable functions. We can write a while program that computes \( f \circ g \).

We can construct computable functions from computable functions. This is because we can build while programs from other while programs.
Examples of Computable Functions

\[ f(x) = 1 \] is computable using \( x := 1 \)

\[ f(x) = x + 1 \] is computable using \( x := x + 1 \)

\[ f(x) = \begin{cases} -x & \text{if } x < 0 \\ x & \text{otherwise} \end{cases} \] is computable using if \( x < 0 \) then \( x := -x \)

Suppose \( f \) and \( g \) are computable functions. We can write a while program that computes \( f \circ g \). Let \( F \) and \( G \) be programs that compute \( f \) and \( g \) then \( G;F \) computes \( f \circ g \).

We can construct computable functions from computable functions. This is because we can build while programs from other while programs.
But what about the function

\[ f(x) = \begin{cases} 
  x^2 & \text{if } x \geq 0 \\
  \text{undefined} & \text{otherwise}
\end{cases} \]

This is computable if we can write a program \( S \) such that

\[ \langle S, [x \mapsto n] \rangle \Rightarrow^* [x \mapsto n^2] \text{ if } n \geq 0. \]

This is easy:

\[ x := x \times x \]

However, this computes the total function \( f(x) = x^2 \). We really want the output of the program to be \textit{undefined} when \( n < 0 \). But what does \textit{undefined} look like for programs? \textit{non-termination}.

\[ \text{if } x \geq 0 \text{ then } x := x \times x \text{ else } (\text{while true do skip}) \]
Computable Functions $\mathbb{N}^m \to \mathbb{N}^n$

Recall our pairing bijection $\phi : (\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$.

**Definition**

A function $f : \mathbb{N}^m \to \mathbb{N}^n$ – with $n, m \geq 1$ – is computable if, and only if, there is a function $g : \mathbb{N} \to \mathbb{N}$ which is computable in the sense of the previous Definition, such that

$$
g(\phi(x_1, \phi(x_2, \ldots \phi(x_{n-1}, x_n)))) = 
(\phi(y_1, \phi(y_2, \ldots \phi(y_{m-1}, y_m) \ldots ))
$$

where $\phi$ is the pairing bijection, and,

$$f(x_1, x_2, \ldots, x_{n-1}, x_n) = (y_1, y_2, \ldots y_{m-1}, y_m)$$
Pigeon Hole Principle

This is a principle that all computer scientists should be familiar with. For finite sets it can be stated as:

*If* $n$ *items are put into* $m$ *containers, with* $n > m$, *then at least one container must contain more than one item*

For infinite sets we need to use more formal language to say

*There does not exist an injective function whose co-domain is smaller than its domain*

but the idea is the same.
The general idea:

1. There are a countable number of while programs
2. There are an uncountable number of functions

Therefore, there must be functions with no corresponding while program, so (by definition) those functions are not computable.

We can replace while by any language and just need to show that programs in that language of countable.

We now show (1) and (2) from above.
Effectively countable and Gödel numbers (from last time)

**Theorem**

The set of while programs is effectively countably infinite.

Effectively means we can write a program that could write them down in a list (enumerate them) and, conversely, find the index for each program.

**Definition**

To obtain the index, code number or Gödel number of a while program $S$, we will use $\gamma(S) = \phi_S(S)$ (the type of $\gamma$ is $\text{Stmt} \to \mathbb{N}$).

We now have

- a function to turn while programs into natural numbers, and
- a function to turn natural numbers into while programs i.e. $\gamma^{-1}$
The set $\mathbb{R}$ is Uncountable

We can show this via Cantor’s Diagonalisation Method.
Uncountable Functions

There are uncountably many functions from \( \mathbb{N} \to \mathbb{N} \).

We will use Cantor’s Diagonalisation Method to prove this.

Assume that there are countably many functions from \( \mathbb{N} \to \mathbb{N} \) and therefore we can enumerate them

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_0 )</td>
<td>( f_0(0) )</td>
<td>( f_0(1) )</td>
<td>( f_0(2) )</td>
<td>( f_0(3) )</td>
<td>( ... )</td>
<td></td>
</tr>
<tr>
<td>( f_1 )</td>
<td>( f_1(0) )</td>
<td>( f_1(1) )</td>
<td>( f_1(2) )</td>
<td>( f_1(3) )</td>
<td>( ... )</td>
<td></td>
</tr>
<tr>
<td>( f_2 )</td>
<td>( f_2(0) )</td>
<td>( f_2(1) )</td>
<td>( f_2(2) )</td>
<td>( f_2(3) )</td>
<td>( ... )</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

The function \( f_{\text{new}}(n) = f_n(n) + 1 \) must be different from every \( f_k \)

The enumeration is incomplete; we cannot count functions in \( \mathbb{N} \to \mathbb{N} \)
Uncountable Functions

There are uncountably many functions from $\mathbb{N} \to \mathbb{N}$.

We will use Cantor’s Diagonalisation Method to prove this.

Assume that there are countably many functions from $\mathbb{N} \to \mathbb{N}$ and therefore we can enumerate them

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_0$</td>
<td>$f_0(0)$</td>
<td>$f_0(1)$</td>
<td>$f_0(2)$</td>
<td>$f_0(3)$</td>
<td>...</td>
<td>$f_{\text{new}}(0) \neq f_0(0)$</td>
</tr>
<tr>
<td>$f_1$</td>
<td>$f_1(0)$</td>
<td>$f_1(1)$</td>
<td>$f_1(2)$</td>
<td>$f_1(3)$</td>
<td>...</td>
<td>$f_{\text{new}}(1) \neq f_1(1)$</td>
</tr>
<tr>
<td>$f_2$</td>
<td>$f_2(0)$</td>
<td>$f_2(1)$</td>
<td>$f_2(2)$</td>
<td>$f_2(3)$</td>
<td>...</td>
<td>$f_{\text{new}}(2) \neq f_2(2)$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

The function $f_{\text{new}}(n) = f_n(n) + 1$ must be different from every $f_k$

The enumeration is incomplete; we cannot count functions in $\mathbb{N} \to \mathbb{N}$
We have defined the notion of **computable functions**

We have proved the existence of **uncomputable functions** in $\mathbb{N} \rightarrow \mathbb{N}$

This result can be lifted to all relevant functions by coding arguments

We have done this in the context of **while** programs but the same arguments can be lifted to any programming language. As programs are necessarily finite, we can count them.

It turns out that the set of computable functions is the **same** for all realistic programming languages

Therefore, the notion of computable function is independent of the **while** language
A model of computation is universal if it can model itself.

By this we mean we can write a program in the language that can simulate all other programs that can be written in the language.

To define a universal program we are going to use our previous trick of encoding programs as numbers.

The idea will be to take that number as input then use $\gamma^{-1}$ to lookup the relevant program and run it.
The function computed by the program $\gamma(i)$

**Definition**

Suppose that the program $S$ is the value of $\gamma^{-1}(i)$. Then we define $\eta_i$, the function associated with the $i$-th program as follows.

\[
\eta_i(k) = \begin{cases} 
\sigma'(x) & \text{if } \exists n, s'. \text{ with } < S, \sigma[x \mapsto k] > \Rightarrow^n \sigma' \\
\text{undefined} & \text{otherwise}
\end{cases}
\]

The undefined case allows for partial functions

The type of $\eta$ is $\mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$

The type of $\gamma$ is $\text{Stm} \rightarrow \mathbb{N}$

The type of $\gamma^{-1}$ is $\mathbb{N} \rightarrow \text{Stm}$
\(\gamma(i)\) is a program whilst \(\eta_i\) is a function

**Lemma**

*The function \(h : \mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{N})\) defined as \(h(i) = \eta_i\) is not injective.*

**Proof sketch:**
- Suppose \(h(i) = h(j)\). If \(h\) were injective then \(i = j\).
- Let \(S = \gamma^{-1}(i)\) and let \(S' = \text{skip};\ S\)
  \[
  \gamma(\text{skip};\ S) = 2 + 4(2\gamma(\text{skip})(2\gamma(S) + 1) - 1)
  \]
  \[
  = 2 + 4((2i + 1) - 1)
  \]
  \[
  = 2 + 8i
  \]
- As \(8i + 2 \neq i\) we do not have \(i = j\).

Each index \(i\) identifies a unique program but each function is computed by an infinite number of programs.
Universal Function

Definition

We define the *Universal Function* $\psi_U : (\mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{N}$ for unary computable functions as:

$$\psi_U(a, b) = \eta_a(b)$$

where $a$ is the code number of a program computing function $\eta_a$.

(the U in $\psi_U$ is for Universal)
The Universal Program computes the Universal Function

The Universal Function $\psi_U(i, x)$ is computable

1. Find the while program associated with index $i$, which is $P_i = \gamma^{-1}(i)$. This is computable.

2. Next we simulate the execution of the program $P_i$, step-by-step. To do this we must code the reduction transitions using natural numbers. As while programs are numbers, these transitions are arithmetic operations, which are computable.

3. If and when the computation stops, the result will be in held in the value of variable $x$ in the final state.

The above steps describe the universal program

The definition and proof can be lifted to n-ary computable functions.
Some things we should agree on:

- We can simulate all `while` programs using a single `while` program
- We can simulate all Java programs using a single Java program
- We can simulate all `while` programs using a Java program
- We can translate any Java program into an equivalent `while` program
- We can simulate all Java programs using a `while` program
- If we have a simulation in each direction then this means that the models of computation are equivalent

**Thesis (Church-Turing)**

*Any sensible definition of computation will define the same functions to be computable as any other definition.*
Take Home Points

A function is computable if we can write a program to compute it.

There are an uncountable number of functions

As there are countable programs there are functions we cannot compute

The \texttt{while} language is universal; it can implement the universal function

The Church-Turing Thesis says that all reasonable models of computation are the same - if a function is computable then we can write a program in \textit{any} language to compute it.