Do you remember?

- States are **total functions** from variables to integers
- The (operational) semantics of **while** programs is captured **unambiguously** by the rewrite relation $\rightarrow$
- **while** programs do not necessarily terminate

Something to think about:

- Which sets are the same size? $\mathbb{N}$, $\mathbb{R}$, $\mathbb{Z}$, $(\mathbb{N} \times \mathbb{N})$, $(\mathbb{N} \rightarrow \mathbb{N})$, $(\mathbb{B} \rightarrow \mathbb{N})$
- Can you write quicksort in **while**? What about binary tree search?
- How many **while** programs are there?
- How many **while** programs are there that implement $f(x) = x$?

It is expected that you read your email. Not reading it is never an excuse.
What do \texttt{while} programs compute?

Answer: Functions in $\mathbb{Z}^n \rightarrow \mathbb{Z}^m$

Because they take $n$ variables with given values and set $m$ variables.

Next week we see that we cannot compute all functions.

Today we consider two points:

1. \textbf{Is this too much?} Do we really need to consider this general notion of \texttt{while} program when we discuss computability next week? The answer is no. We can show that considering functions in $\mathbb{N} \rightarrow \mathbb{N}$ is sufficient by coding functions in $\mathbb{Z}^n \rightarrow \mathbb{Z}^m$ into $\mathbb{N} \rightarrow \mathbb{N}$.

2. \textbf{Is this enough?} Could we increase the expressiveness of the \texttt{while} language by extending it with more kinds of structures (such as lists, strings, classes etc)? Again, the answer is no. We show this by coding functions in arbitrary (countable) domains in $\mathbb{N} \rightarrow \mathbb{N}$. 
From $\mathbb{Z}^n \to \mathbb{Z}^m$ to $\mathbb{N} \to \mathbb{N}$ (and back again)

Today we will show that

1. We can directly relate functions in $\mathbb{Z} \to \mathbb{Z}$ and functions in $\mathbb{N} \to \mathbb{N}$
2. We can lift this to arbitrary sequences i.e. pairs, triples, lists

and thus, any function in $\mathbb{Z}^n \to \mathbb{Z}^m$ can be represented by one in $\mathbb{N} \to \mathbb{N}$.

I use the term relate as we will introduce bijections meaning that we can go in either direction.
To answer the good enough question we lift our arguments.

The idea is to show that all functions $A \rightarrow B$ can be mapped to some function in $\mathbb{N} \rightarrow \mathbb{N}$ for any countable domains $A$ and $B$.

For example

- if $A$ were the set of all integer arrays and $B$ was integers then we could capture the max function on arrays
- if $A$ and $B$ were both arrays then we could capture sorting
- if $A$ were the set of graphs and $B$ the set of node lists then we could capture Dijkstra’s algorithm

To do this we will argue that we can generalise our approach to code other domains into $\mathbb{N}$
Mapping $\mathbb{Z}$ to $\mathbb{N}$

Lemma

There is a bijection between $\mathbb{N}$ and $\mathbb{Z}$

Let us define $\beta(x) : \mathbb{Z} \rightarrow \mathbb{N}$ as

$$
\beta(x) = \begin{cases} 
2x & \text{if } x \geq 0 \\
-2x - 1 & \text{otherwise}
\end{cases}
$$

For example,

$$
\beta(0) = 0 \quad \beta(1) = 2 \quad \beta(-1) = 1 \quad \beta(5) = 10 \quad \beta(-3) = 5
$$

i.e. it maps positive numbers to even numbers and negative to odd

This is a bijection
We can compute $\beta$ 

```plaintext
if $x \geq 0$ then $z := 2 \times x$ else $z := (-2 \times x) - 1$
```

We can compute its inverse $\beta^{-1}$ (on inputs in $\mathbb{N}$) 

```plaintext
r := x; z := 0;
while ($2 \leq r$) do ($z := z+1; r = r-2$);
if $r=1$ then $z := -z - 1$
```

(Should probably prove that it is an inverse)

We can transform any program using $\mathbb{Z}$ into one using $\mathbb{N}$ only
Dealing with Pairs

Next we show that we can code arbitrary pairs of numbers as single numbers (in \( \mathbb{N} \) due to the previous argument).

The following function \( \phi \) maps pair \((n, m)\) to a unique number in \( \mathbb{N} \):

\[
\phi(n, m) = 2^n(2m + 1) - 1
\]

<table>
<thead>
<tr>
<th>m</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>11</td>
<td>19</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

\[\phi(2, 3) = 27 = 11011b\]
\[\phi(3, 4) = 71 = 1000111b\]

The function \( \phi \) is a bijection - see notes for proof.
Lists (of integers) can be recursively defined as

\[
\text{list} = [] \mid n :: \text{list}
\]

(Similar but syntax to COMP11120)

We can code lists using a recursively defined function \( \varphi \)

\[
\varphi([]) = 0 \\
\varphi(n :: l) = 2^n(2\varphi(l) + 1) = \phi(n, \varphi(l)) + 1
\]

The +1 at the end is to give 'space' for the empty list

This trick of using \( \phi \) to define \( \varphi \) means we just need a good \( \phi \)
(Not Examinable)

What if we wanted to code pairs such that \( \rho(a, b) = \rho(b, a) \)?

We need to play some games as clearly \( \rho : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \) satisfying this will not be a bijection. Why?

If we have such a \( \rho \) can we lift this to code arbitrary sets?
Exercise

a. Define a bijective function between a triple of natural numbers and the natural numbers e.g. $\phi_3 : (\mathbb{N} \times \mathbb{N} \times \mathbb{N}) \to \mathbb{N}$. You may find it useful to use $\phi$.

b. Show that your function $\phi_3$ is a bijection.

c. Define a coding function that codes a binary tree as a natural number. Recall that a binary tree can be defined recursively as

$$btree = empty \mid (n, btree, btree)$$

for $n \in \mathbb{N}$. You may find it useful to use $\phi_3$. 
What is missing in the above is the point that operations on coded data structures become arithmetic.

We can write a while program to

- construct a pair
- extract one half of a pair
- extract the head or tail of a list
- append two lists
- reverse a list

etc

All using the arithmetic operations available in while.
Everything can be written down as a list or a pair e.g.

- An object in Java is a list of the fields that it contains
- Characters can be mapped directly to numbers
- Strings are just lists of characters
- A graph is a list of pairs

So we can just deal with programs that compute functions in $\mathbb{N} \rightarrow \mathbb{N}$ and argue that we can also compute functions on any other data structures
What about while programs themselves?

The previous argument was that if we have a countable domain then we can code it into $\mathbb{N}$ and hence represent it using while programs.

Can we represent while programs themselves in this way?

Yes if we have a countable number of while programs.

Next we show that the set of while programs is countably infinite.
We can define a bijection $\phi_S : \textbf{Stm} \rightarrow \mathbb{N}$ to code statements in while into $\mathbb{N}$. All the details are in the notes and we just sketch the idea here.

Recall that the syntax of the while language is

\[
\begin{align*}
S & ::= \ x := a \mid \text{skip} \mid S_1; S_2 \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \mid \text{while } b \text{ do } S \\
\ b & ::= \ \text{true} \mid \text{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \\
\ a & ::= \ x \mid n \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2
\end{align*}
\]

To define $\phi_S$ we need a bijection that takes any $S$ and produces a natural number. First we need to deal with arithmetic and boolean expressions.
Coding Arithmetic Expressions

We code each of the cases

\[ \phi_A : \text{AExp} \to \mathbb{N} \]

\[ \phi_A(n) = 5 \times n \]
\[ \phi_A(x) = 1 + 5 \times x \]
\[ \phi_A(a_1 + a_2) = 2 + 5 \times \phi(\phi_A(a_1), \phi_A(a_2)) \]
\[ \phi_A(a_1 - a_2) = 3 + 5 \times \phi(\phi_A(a_1), \phi_A(a_2)) \]
\[ \phi_A(a_1 \times a_2) = 4 + 5 \times \phi(\phi_A(a_1), \phi_A(a_2)) \]

What do we need to do for \( n \) and \( x \)?
Coding Arithmetic Expressions

We code each of the cases

\[ \phi_A : \text{AExp} \to \mathbb{N} \]

\[ \phi_A(n) = 5 \times n \]
\[ \phi_A(x) = 1 + 5 \times x \]
\[ \phi_A(a_1 + a_2) = 2 + 5 \times \phi(\phi_A(a_1), \phi_A(a_2)) \]
\[ \phi_A(a_1 - a_2) = 3 + 5 \times \phi(\phi_A(a_1), \phi_A(a_2)) \]
\[ \phi_A(a_1 \times a_2) = 4 + 5 \times \phi(\phi_A(a_1), \phi_A(a_2)) \]

What do we need to do for \( n \) and \( x \)? Create a bijection with \( \mathbb{N} \)
We code each of the cases

\[ \phi_A : \text{AExp} \rightarrow \mathbb{N} \]
\[ \phi_A(n) = 5 \times n \]
\[ \phi_A(x) = 1 + 5 \times x \]
\[ \phi_A(a_1 + a_2) = 2 + 5 \times \phi_A(a_1, \phi_A(a_2)) \]
\[ \phi_A(a_1 - a_2) = 3 + 5 \times \phi_A(a_1, \phi_A(a_2)) \]
\[ \phi_A(a_1 \times a_2) = 4 + 5 \times \phi_A(a_1, \phi_A(a_2)) \]

What do we need to do for \( n \) and \( x \)? Create a bijection with \( \mathbb{N} \)

Why does this work? (you need to understand for the exercises)

Example: \( \phi_A(1 + 1) = 2 + 5\phi_A(5, 5) = 2 + 5(2^5(2 \times 5 + 1) - 1) = 1762 \)

1762 = 2 + 5x (for x = 352), all numbers of the form 2 + 5x hold a \( a_1 + a_2 \)
true and false only need one 'space' each. Note how we use $\phi$ and $\phi_A$.

$\phi_B : \mathbf{BExp} \rightarrow \mathbb{N}$

$\phi_B(\text{true}) = 0$
$\phi_B(\text{false}) = 1$
$\phi_B(a_1 = a_2) = 2 + 4 \times \phi(\phi_A(a_1), \phi_A(a_2))$
$\phi_B(a_1 \leq a_2) = 3 + 4 \times \phi(\phi_A(a_1), \phi_A(a_2))$
$\phi_B(\neg b) = 4 + 4 \times \phi_B(b)$
$\phi_B(b_1 \land b_2) = 5 + 4 \times \phi(\phi_B(b_1), \phi_B(b_2))$
Extending the idea to Statements

Exercise

Using φ, φ_A and φ_B, complete the following recursive function φ_S which is a bijection from Stm to N:

\[
\begin{align*}
\phi_S : \text{Stm} & \to \mathbb{N} \\
\phi_S(\text{skip}) & = 0 \\
\phi_S(\text{while } b \text{ do } S) & = 1 + 4 \times \phi(\phi_B(b), \phi_S(S)) \\
\phi_S(x := a) & = ? \\
\phi_S(S_1; S_2) & = ? \\
\phi_S(\text{if } b \text{ then } S_1 \text{ else } S_2) & = ?
\end{align*}
\]

Argue that this means there are only countably many programs in while.
Effectively countable and Gödel numbers

**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. $\gamma^{-1}$
Take Home Points

Any function in $\mathbb{Z}^n \rightarrow \mathbb{Z}^m$ can be coded into a function in $\mathbb{N} \rightarrow \mathbb{N}$.

This means that we can restrict our attention to functions in $\mathbb{N} \rightarrow \mathbb{N}$ and the while programs that compute them and any results will generalise.

We have bijections between $\mathbb{N}$ and $\mathbb{Z}$, $(\mathbb{N} \times \mathbb{N})$ and $List(\mathbb{N})$ (or $\mathbb{N}^n$)

There are countably infinite many while programs (we have a bijection between while programs and $\mathbb{N}$)

We can forget the implementation details of the bijection and refer to the Gödel number of a program (we do this next week)