A Universal Register Machine.
Part I:
Coding register machines as numbers

A key part of the Turing/Church solution of Hilbert's Entscheidungs-
problem was to exploit the idea that
(formal descriptions of) algorithms can be the data
on which algorithms act.

We are using register machines as the formal description of the informal
notion of "algorithm". Since the data that register machines manipulate
are numbers, to develop the above idea we have to [have an
algorithm to] code register machines as numbers.*

To do that we need to be able to code pairs of numbers
finite lists of numbers
as numbers. There are many ways of doing that: we fix upon one
convenient way...

* such codings are often called Gödel numberings, after Gödel's original
use of the idea: his coding of arithmetic formulas as numbers was
a key part of his proof of the famous Incompleteness Theorem.
Coding pairs of numbers as numbers

For \( x, y \in \mathbb{N} \) define

\[
\langle x, y \rangle \ \overset{\text{def}}{=} \ 2^x(2y+1)
\]

\[
\langle x, y \rangle \ \overset{\text{def}}{=} \ 2^x(2y+1) - 1
\]

Thus

\[
\langle x, y \rangle \ \text{in binary} = \begin{array}{c} y \ \text{in binary} \\ 0 \cdots 0 \end{array}
\]

\[
\langle x, y \rangle \ \text{in binary} = \begin{array}{c} y \ \text{in binary} \\ 1 \cdots 1 \end{array}
\]

\[
x \ 0's
\]

\[
x \ 1's
\]

Hence

- \( \langle -,- \rangle \) and \( \langle -,- \rangle \) both determine injective functions from \( \mathbb{N} \times \mathbb{N} \) to \( \mathbb{N} \), i.e.

\[
\langle x_1, y_1 \rangle = \langle x_2, y_2 \rangle \Rightarrow x_1 = x_2 \text{ and } y_1 = y_2
\]

\[
\langle x_1, y_1 \rangle = \langle x_2, y_2 \rangle \Rightarrow x_1 = x_2 \text{ and } y_1 = y_2
\]

- \( \langle -,- \rangle \) is a surjective function from \( \mathbb{N} \times \mathbb{N} \) to \( \{ z \in \mathbb{N} | z \neq 0 \} \), i.e. for all \( z \neq 0 \) there are \( x, y \in \mathbb{N} \) with \( \langle x, y \rangle = z \).

- \( \langle -,- \rangle \) is a surjective function from \( \mathbb{N} \times \mathbb{N} \) to \( \mathbb{N} \), i.e. for all \( z \) there are \( x, y \in \mathbb{N} \) with \( \langle x, y \rangle = z \).

and hence \( \langle -,- \rangle \) is a bijection (aka. one-to-one correspondence) between \( \mathbb{N} \times \mathbb{N} \) and \( \mathbb{N} \).

(and \( \langle -,- \rangle \) a bijection between \( \mathbb{N} \times \mathbb{N} \) and \( \{ z | z \neq 0 \} \).
NOTATION for lists (of numbers)

\[ \mathbb{N}^* \overset{\text{def}}{=} \text{set of finite lists of natural numbers} \]

\[ = \{ \text{nil} \} \cup \mathbb{N} \cup \mathbb{N}^2 \cup \ldots \cup \mathbb{N}^n \cup \ldots \]

unique list of length 0 lists of length 1 lists \((x_1, \ldots, x_n)\) of length \(n\)

\(\text{cons} \in \text{Fun}(\mathbb{N} \times \mathbb{N}^*, \mathbb{N}^*)\)

\(\text{head} \in \text{Pfn}(\mathbb{N}^*, \mathbb{N})\)

\(\text{tail} \in \text{Fun}(\mathbb{N}^*, \mathbb{N}^*)\)

cons is a bijection from \(\mathbb{N} \times \mathbb{N}^*\) to \(\{ l \in \mathbb{N}^* | l \neq \text{nil} \}\)

\[
\begin{align*}
\text{head}(\text{cons}(x, l)) &= x \\
\text{head}(\text{nil}) &= \uparrow \\
\text{tail}(\text{cons}(x, l)) &= l \\
\text{tail}(\text{nil}) &= \text{nil} \\
l &= \text{cons}(\text{head}(l), \text{tail}(l)) \text{ if } l \neq \text{nil}
\end{align*}
\]

Every list can be built up from \text{nil} by repeated \text{cons}’s:

\[(x_1, \ldots, x_n) = \text{cons}(x_1, \text{cons}(x_2, \ldots, \text{cons}(x_n, \text{nil}) \ldots))\]

Coding lists \((x_1, \ldots, x_n) \in \mathbb{N}^*\) as numbers \([x_1, \ldots, x_n] \in \mathbb{N}\)

Define \([x_1, \ldots, x_n] \in \mathbb{N}\) by induction on the length of the list \((x_1, \ldots, x_n) \in \mathbb{N}^*\):

CASE \(n = 0\) : \([\text{nil}] \overset{\text{def}}{=} 0\)

INDUCTION STEP : \([\text{cons}(x, l)] \overset{\text{def}}{=} \langle x, [l] \rangle = 2^x(2^|l| + 1)\)

Thus in general \([x_1, \ldots, x_n] = \langle x_1, \langle x_2, \ldots, \langle x_n, 0 \rangle \ldots \rangle \rangle\)

From the definition of \(\langle - , - \rangle\) we get :

\[
[x_1, \ldots, x_n] \text{ in binary} = 100000 \ldots 100000
\]

number of 1’s in \(x_n\) 0’s \(x_{n-1}\) 0’s \(x_1\) 0’s

Hence \(l \mapsto [l]\) determines a bijection from \(\mathbb{N}^*\) to \(\mathbb{N}\)
Examples

\[ [3] = [\text{cons}(3, \text{nil})] = (3, 0) = 2^3(2.0 + 1) = 8 \text{ decimal} \]

\[ = \frac{1000}{3} \text{ binary} \]

\[ [1, 3] = (1, [3]) = (1, 8) = 2^4(2.8 + 1) = 34 \text{ decimal} \]

\[ = \frac{100010}{3} \frac{1}{3} \text{ binary} \]

\[ [2, 1, 3] = (2, [1, 3]) = (2, 34) = 2^2(2.34 + 1) = 276 \text{ decimal} \]

\[ = \frac{10010100}{3} \frac{1}{3} \frac{2}{3} \text{ binary} \]

(NB reversal of order of list when reading left-to-right in binary representation.)
Coding register machine programs $\text{Prog}$ numbers $\langle \text{Prog} \rangle \in \mathbb{N}$

If $\text{Prog}$ is $\text{LO} : \text{body}_0$
$L_1 : \text{body}_1$
$\vdots$
$L_m : \text{body}_m$

then $\langle \text{Prog} \rangle \overset{\text{def}}{=} [\text{code}(\text{body}_0), \ldots, \text{code}(\text{body}_m)]$

\[
\begin{align*}
\text{code}(\text{Ri}^+ \rightarrow L_j) & \overset{\text{def}}{=} \langle 2i, j \rangle \\
\text{code}(\text{Ri}^- \rightarrow L_j, L_k) & \overset{\text{def}}{=} \langle 2i + 1, \langle j, k \rangle \rangle \\
\text{code}(\text{HALT}) & \overset{\text{def}}{=} 0
\end{align*}
\]

Any $x \in \mathbb{N}$ decodes uniquely as an instruction:

- if $x = 0$ then the instruction is $\text{HALT}$
- else decode $x$ as a pair $x = \langle y, z \rangle$ and
  - if $y$ is even then instruction is $\text{Ri}^+ \rightarrow L_j$ where $i = y/2$, $j = z$
  - else ($y$ is odd and) decode $z$ as a pair $z = \langle u, v \rangle$ and then the instruction is $\text{Ri}^- \rightarrow L_j, L_k$ where $i = (y-1)/2$, $j = u$ and $k = v$.

Hence any $e \in \mathbb{N}$ decodes uniquely as a program $\text{Pro}_{e}$ called the (register machine) program with index $e$:

- first decode $e$ as a list $e = [x_1, \ldots, x_n]$
- and then decode each $x_i$ as an instruction, as above.
(1) The program resulting from this decoding process may well have jumps to labels greater than the length of the list of instructions, i.e. the associated register machine may well be capable of halting erroneously— but no matter.

(2) In case $e = 0 = [\text{nil}]$ we get an empty list of instructions, which by convention we regard as a machine that does nothing.

Example: decode 666 as a program (for the register machine from hell!)

decimal 666 = binary 1010011010

= [1, 1, 0, 2, 1]

Now

0 is code for instruction HALT

$1 = \langle 0, 0 \rangle$ is code for instruction $R0^+ \rightarrow L0$

$2 = \langle 1, 0 \rangle = \langle 1, \langle 0, 0 \rangle \rangle$ is code for $R0^- \rightarrow L0, L0$

So 666 decodes to the program

$\begin{align*}
L0 & : R0^+ \rightarrow L0 \\
L1 & : R0^+ \rightarrow L0 \\
L2 & : \text{HALT} \\
L3 & : R0^- \rightarrow L0, L0 \\
L4 & : R0^+ \rightarrow L0
\end{align*}$

(which never halts).

A Universal Register Machine.

Part II: Description of the machine.
High-level description of a universal register machine, \( U \):

- \( U \) has registers \( P(\text{rogram}), A(\text{rgument}), \ldots \)
- Loading \( P \) with value \( e \), \( A \) with value \( a \) and all other registers with \( 0 \), then \( U \) acts as follows:
  - decode \( e \) as a program: \( e = \text{Prog}_e \)
  - decode \( a \) as a list of register values: \( a = [a_1, \ldots, a_n] \)
  - carry out the computation of the register machine program \( \text{Prog}_e \) starting with registers \( R_0, R_1, \ldots, R_n \) set to \( 0, a_1, a_2, \ldots, a_n \) (and any other registers occurring in \( \text{Prog}_e \) set to \( 0 \)).

Overall structure of \( U \)'s program:

1. copy \( P \) to \( T \), copy \( PC^{th} \) item of list in \( T \) to \( N \) (HALTING if \( PC > \text{length of list} \)) and goto 2

2. if \( N = 0 ( = \text{code(HALT)} ) \) then HALT, else decode \( N \) as \( \langle y, z \rangle \), assign \( y \) to \( C \) & \( z \) to \( N \), and goto 3

   \{ At this point either \( C = 2i \) is even & current instruction is \( R_i^{+} \rightarrow L_j \) where \( j = N \) or \( C = 2i + 1 \) "odd" \( " \text{" } R_i \rightarrow L_j; L_k \) \( \langle j, k \rangle = N \}\}

3. remove register values from list in \( A \) up to the one required \{ the \( i^{th} \} \), putting it in \( R \) & saving preceding values as a list in \( S \), then goto 4

4. execute current instruction on value in \( R \), update \( PC \) \{ to \( j \) or \( k \) as above \}, restore register values from \( R \) and \( S \) to \( A \), and goto 1

* see Note on p 42
The registers of U and the role they play in its program:

- **P** - holds the code of the register machine to be simulated
- **A** - holds current contents of registers of the register machine being simulated
- **PC** - program counter - holds the number of the current instruction
  (counting from 0)
- **N** - holds the code of the current instruction
- **C** - indicates the type of the current instruction
- **R** - holds the contents of simulated machine's register that is to be
  incremented/decremented by current instruction (if not a HALT instruction)
- **T** - holds a working copy of the program code
- **S, Z** - auxiliary registers for intermediate computations

---

**Note**

At step 3 it may be that \( i > \text{length of the list in } A \),
i.e. that current instruction wants to increment/decrement a register
in the simulated machine that was not assigned a value by the
initial value of \( A \) or has not been assigned so far in the
interpreted program. By assumption, such a register has value 0.
So in this case, say \( A = [a_1, \ldots, a_n] \) with \( i > n \), at step 3
\( A \) will be set to \( 0 (= [\text{nil}] \) ), \( R \) set to 0, and \( S \) set to
\([0, \ldots, 0, a_n, \ldots, a_i]\). Then when the register values are restored
\[
\begin{align*}
\text{at step 4, } A \text{ will hold } & [a_1, \ldots, a_n, 0, \ldots, 0, r] \\
& \text{where } r \text{ is the value in } R \text{ after executing the current instruction.}
\end{align*}
\]

The detailed construction of U's program depends on the fact that
various procedures for manipulating (codes of) lists of numbers
are register machine programmable...
The program to carry out \( S := R \) can be implemented by

Precondition:
\[
\begin{align*}
R &= x \\
S &= y \\
Z &= 0
\end{align*}
\]

Postcondition:
\[
\begin{align*}
R &= x \\
S &= x \\
Z &= 0
\end{align*}
\]
The program to carry out \((X, L) := (0, \text{cons}(X, L))\) can be implemented by

\[
\begin{align*}
\text{Precondition:} & \quad \begin{align*}
X &= x \\
L &= l \\
Z &= 0
\end{align*} \\
\text{Postcondition:} & \quad \begin{align*}
X &= 0 \\
L &= \langle x, l \rangle = 2^x(2l+1) \\
Z &= 0
\end{align*}
\end{align*}
\]
The program to carry out \((X, L) := (0, \text{cons}(X, L))\) can be implemented by

\begin{align*}
(L, Z) &:= (2L + Z, 0) \\
\text{Precondition :} & \\
X &= X \\
L &= L \\
Z &= 0 \\
\text{Postcondition :} & \\
X &= 0 \\
L &= \langle x, l \rangle = 2^x(2l+1) \\
Z &= 0
\end{align*}

\begin{align*}
(L, Z) &:= (2L + Z, 0) \\
(L, Z) &:= (2L + 1 + Z, 0) \\
\text{Precondition :} & \\
X &= x \\
L &= l \\
Z &= 0 \\
\text{Postcondition :} & \\
X &= 0 \\
L &= \langle x, l \rangle = 2^x(2l+1) \\
Z &= 0
\end{align*}
The program

```
START \rightarrow \text{pop } L \rightarrow x \rightarrow \text{HALT}
\downarrow
\text{EXIT}
```

Specification:

```
\text{START} \rightarrow L = \text{nil} ?
\rightarrow \text{no} \rightarrow (x, L) := (\text{head}(L), \text{tail}(L)) \rightarrow \text{HALT}
\rightarrow \text{yes} \rightarrow \text{X} := 0 \rightarrow \text{EXIT}
```

"if \( L = \text{nil} \) then assign \( 0 \) to \( X \) and goto EXIT,
else \( L = (x, l) \) say, assign \( x \) to \( X \) and \( l \) to \( L \),
and goto \( \text{HALT} \)"

Implementation of

```
\text{START} \rightarrow \text{pop } L \rightarrow x \rightarrow \text{HALT}
\downarrow
\text{EXIT}
```

```
\text{START} \rightarrow X^- \rightarrow L^- \rightarrow L^+ \rightarrow L^- \rightarrow Z^- \rightarrow Z^- \rightarrow \text{HALT}
\rightarrow \text{EXIT}
```
Implementation of

START $\rightarrow$ pop L to X $\rightarrow$ HALT

(assuming $Z=0$, $L>0$)
(while $L$ even do
($L := \frac{1}{2}L$ ; $X := X+1$);
$L := \frac{1}{2}(L-1)$

if $Z+L$ even then
$(Z,L) := (0, \frac{1}{2}(Z+L))$ & goto E
else
$(Z,L) := (0, \frac{1}{2}(Z+L-1))$ & goto 0

START $\rightarrow$ X $\rightarrow$ L $\rightarrow$ L $\rightarrow$ L $\rightarrow$ E $\rightarrow$ 0 $\rightarrow$ HALT

The program for $U$:

START $\rightarrow$ copy P to T $\rightarrow$ pop T to N $\rightarrow$ HALT

1

2

3

4

HALT $\rightarrow$ pop N to C

pop A to R

pop S to R

push R to A

Copy N to PC

R $\rightarrow$ R $\rightarrow$ N $\rightarrow$ C $\rightarrow$ push R to S

C even

C odd