1. The Universality Theorem
2. The Step-Counter Theorem
Universality Theorem:

**Theorem**

For each $n > 0$ there exists a partially computable function $\Phi^{(n)}$ such that if $\mathcal{P}$ is an $S$-program with $\#(\mathcal{P}) = y$, then we have:

$$\Phi^{(n)}(x_1, \ldots, x_n, y) = \psi^{(n)}_{\mathcal{P}}(x_1, \ldots, x_n)$$
Universal program $U_n$ acts like an interpreter for computable functions of $n$ arguments.
The proof of the theorem consists in the construction of a program $\mathcal{U}_n$ for each $n > 0$ such that

$$\psi_{\mathcal{U}_n}^{(n+1)}(x_1, \ldots, x_n, x_{n+1}) = \Phi^{(n)}(x_1, \ldots, x_n),$$

when $x_{n+1}$ is the code of a program that computes $\Phi^{(n)}$. The program $\mathcal{U}_n$ is called *universal*. It must

- keep track of the current snapshot of $\mathcal{P}$, and
- by decoding the number of the program being interpreted decide what to do next and do it.
Encoding the state of program $\mathcal{P}$ in a variable $S$:
If the $i^{\text{th}}$ variable in the list of variables has the value $a_i$ and all
variables after the $m^{\text{th}}$ variables have value 0, the encoding of the
state is $[a_1, \ldots, a_m]$.

Example

The state $Y = 0, X_1 = 2, X_2 = 1$ is encoded as

$$[0, 2, 0, 1] = 3^2 \cdot 7 = 63.$$  

Note that the input variables occupy even numbered positions in
the list of variables.
The variable $K$ contains the number that indicates the number of the instruction about to be executed.

Recall that the program $U_n$ will compute

$$Y = \Phi^{(n)}(X_1, \ldots, X_n, X_{n+1}),$$

where $X_{n+1} = \#(P)$. The beginning of $U$ consists of

$$Z \leftarrow X_{n+1} + 1$$
$$S \leftarrow \prod_{i=1}^{n} (p_{2i})^{X_i}$$
$$K \leftarrow 1$$

Note: the successive fragments of the program $U_n$ will be shown in this color.
- If $X_{n+1} = \#(\mathcal{P})$, where $\mathcal{P}$ consists of instructions $I_1, \ldots, I_m$, then $Z$ gets the value $[\#(I_1), \ldots, \#(I_m)]$.

- $S$ is initialized at $[0, X_1, 0, X_2, \ldots, 0, X_n]$ which initializes the input variables and sets all other variables to 0.

- $K$ is given the initial value 1 so that the computation can begin with the first instruction.
Next, the line

\[ \text{IF } K = \text{Lt}(Z) + 1 \lor (K = 0) \text{ GOTO } F \]

has the role of determine the end of the computation.
The current instruction must be decoded and executed:

\[
U \leftarrow r((Z)_K)
\]

\[
P \leftarrow p_r(U) + 1
\]

Note that \((Z)_K = \langle a, \langle b, c \rangle \rangle\) is the number of the \(K^{th}\) instruction. Thus, \(U = \langle b, c \rangle\) is the code of the statement about to be executed. The variable mentioned in the \(K^{th}\) instruction is the \(c + 1^{st}\) in the list, that is, \(r(U) + 1^{st}\) in the list. Its current value is stored as the exponent to which \(P\) divides \(S\).
Depending on $b = \ell(U)$ and on the value of $\sim (P|S)$ we continue to certain labels:

- IF $\ell(U) = 0$ GOTO $N$
- IF $\ell(U) = 1$ GOTO $A$
- IF $\sim (P|S)$ GOTO $N$
- IF $\ell(U) = 2$ GOTO $M$
If $\ell(U) = 0$ the instruction is $V \leftarrow V$ and the computation does nothing to $S$.

If $\ell(U) = 1$ the instruction is $V \leftarrow V + 1$, so 1 has to be added to the exponent of $P$ in the prime power factorization of $S$. Then, the computation executes a GOTO $A$.

If $\ell(U)$ is neither 0 nor 1, then the current instruction is either $V \leftarrow V - 1$ or IF $V \neq 0$ GOTO $L$. In either case, if $P$ is not a divisor of $S$, that is, if the current value of $V$ is 0, the computation does nothing to $S$.

If $P|S$ and $\ell(U) = 2$, the computation executes a GOTO $M$ ($M$ for minus), so 1 is subtracted from the exponent to which $P$ divides $S$. 
The program continues with

\[ K \leftarrow \min_{i \leq \ell_t(Z)} [\ell((Z)_i) + 2 = \ell(U)] \]

GOTO C

If \( \ell(U) > 2 \) and \( P \mid S \) the current instruction is

IF \( V \neq 0 \) GOTO L,

where \( V \) has a non-zero value and \( L \) is the label whose position is \( \ell(U) - 2 \). The instruction executed next is the first with this label, so \( K \) should be the least \( i \) such that \( \ell((Z)_i) = \ell(U) - 2 \). If there is no instruction with the appropriate label, \( K \) gets 0, so the program terminates.
In either case, GOTO C causes a jump to the beginning of the loop for the next instruction (if any) to be processed. Next, we have:

\[
\begin{align*}
&M \quad S \leftarrow \lfloor S/P \rfloor \\
&\quad \text{GOTO } N \\
&A \quad S \leftarrow S \cdot P \\
&\quad \text{GOTO } C \\
&N \quad K \leftarrow K + 1 \\
&\quad \text{GOTO } C
\end{align*}
\]

GOTO C causes a jump to the beginning of the loop for the next instruction to be processed.
- $S \leftarrow \lfloor S/P \rfloor$ subtracts 1 from the value of the variable mentioned in the current instruction.
- $S \leftarrow S \cdot P$ adds 1 to the value of the variable mentioned in the current instruction.

The program concludes with

$$[F] \quad Y \leftarrow (S)_1$$
The Program $\mathcal{U}_n$

\begin{align*}
Z & \leftarrow X_{n+1} + 1 \\
S & \leftarrow \prod_{i=1}^{n} (p_{2i})^{X_i} \\
K & \leftarrow 1

[C] & \quad \text{IF } K = \text{Lt}(Z) + 1 \lor (K = 0) \text{ GOTO } F \\
& \quad U \leftarrow r((Z)_K) \\
& \quad P \leftarrow p_{r(U)+1} \\
& \quad \text{IF } \ell(U) = 0 \text{ GOTO } N \\
& \quad \text{IF } \ell(U) = 1 \text{ GOTO } A \\
& \quad \text{IF } \not\sim (P|S) \text{ GOTO } N \\
& \quad \text{IF } \ell(U) = 2 \text{ GOTO } M \\
& \quad K \leftarrow \min_{i \leq \text{Lt}(Z)} [\ell((Z)_i) + 2 = \ell(U)] \\
& \quad \text{GOTO } C

[M] & \quad S \leftarrow \lfloor S/P \rfloor \\
& \quad \text{GOTO } N

[A] & \quad S \leftarrow S \cdot P \\
[N] & \quad K \leftarrow K + 1 \\
& \quad \text{GOTO } C

[F] & \quad Y \leftarrow (S)_1
\end{align*}
On termination, the value of $Y$ is stored as the exponent on $p_1$ (which is 2).
For \( n > 0 \), the sequence

\[
\Phi^{(n)}(x_1, \ldots, x_n, 0), \Phi^{(n)}(x_1, \ldots, x_n, 1), \ldots
\]

enumerates all partially computable functions of \( n \) variables. An alternative notation is

\[
\Phi^{(n)}_y(x_1, \ldots, x_n) = \Phi^{(n)}(x_1, \ldots, x_n, y).
\]

For \( n = 1 \) we use the simplified notation

\[
\Phi_y(x) = \Phi(x, y) = \Phi^{(1)}(x, y).
\]
The Step-Counter Theorem

**Theorem**

Let $\text{STP}^{(n)}$ be the predicate:

$$\text{STP}^{(n)}(x_1, \ldots, x_n, y, t) = \begin{cases} 
1 & \text{if program number } y \text{ halts after } t \text{ or fewer steps on inputs } x_1, \ldots, x_n \\
0 & \text{otherwise.}
\end{cases}$$

For each $n > 0$, $\text{STP}^{(n)}$ is primitive recursive.

Note that $\text{STP}^{(n)}$ is TRUE if there is a computation of program $y$ of length not greater than $t$ beginning with inputs $x_1, \ldots, x_n$. 
The proof operates on numeric versions of the notion of snapshot. If \( z \) represents a state \( \sigma \) of the program \( y \), the number \( \langle i, z \rangle \) represents the snapshot \( (i, \sigma) \). Therefore, for a program \( \mathcal{P} \) whose code is
\[
y = \#(\mathcal{P}) = [(\#(I_1), \#(I_2), \ldots, \#(I_n)] - 1,
\]
the code of instruction \( I_i \) is \( (y + 1)_i \).
Proof cont’d

The following functions extract the components of the $i^{th}$ instruction of the program number $y$, namely, the label, the variable number, the instruction type, and the label to which the $i^{th}$ instruction is pointing:

\[
\begin{align*}
\text{LABEL}(i, y) &= \ell((y + 1)_i), \\
\text{VAR}(i, y) &= r(r((y + 1)_i)) + 1, \\
\text{INSTR}(i, y) &= \ell(r((y + 1)_i)), \\
\text{LABEL}'(i, y) &= \ell(r((y + 1)_i)) \div 2.
\end{align*}
\]
Proof cont’d

Next, we define some predicates that indicate, for program $y$ and snapshot $x$, which kind of action is to be performed:

Recall that if $x$ is a snapshot, $\ell(x)$ is the number of the instruction about to be executed and $r(x)$ represents the state of the program.

$$\text{SKIP}(x, y) \iff \begin{cases} \text{INSTR}(\ell(x), y) = 0 \& \ell(x) \leq \text{Lt}(y + 1) \\ \vee [\text{INSTR}(\ell(x), y) \geq 2 \& (p_{\text{VAR}(\ell(x), y)} | r(x))] \end{cases}$$

This says that if the type of the instruction is $V \leftarrow V$ or the instruction is an IF $V \neq 0 \text{ GOTO } L$ and the value of $V$ is 0 (expressed as $(p_{\text{VAR}(\ell(x), y)} | r(x))$, then the program skips to the next instruction.
Proof cont’d

Proof.

\[
\begin{align*}
\text{INCR}(x, y) & \iff \text{INSTR}(\ell(x), y) = 1 \\
\text{DECR}(x, y) & \iff \text{INSTR}(\ell(x), y) = 2 \& p_{\text{VAR}(\ell(x), y)} | r(x)
\end{align*}
\]

INCR\((x, y)\) is TRUE if the instruction is \(V \leftarrow V + 1\);\nDECR\((x, y)\) is TRUE if the instruction is \(V \leftarrow V - 1\) and the value of \(V\) is not 0;
Proof.

\[
\text{BRANCH}(x, y) \iff \text{INSTR}(\ell(x), y) > 2\& \rho_{\text{VAR}}(\ell(x), y)\mid r(x) \\
\& (\exists i)_{\leq \text{Lt}(y+1)} \text{LABEL}(i, y) = \text{LABEL}'(\ell(x), y).
\]

BRANCH is TRUE if the instruction is of type IF \( V \neq 0 \) GOTO \( L \), the value of the variable \( V \) is not 0 (expressed by \( \rho_{\text{VAR}}(\ell(x), y)\mid r(x) \)), and there exists an instruction with the label \( L \), where the flow may continue.
The function $SUCC(x, y)$ gives the representation of the successor of the snapshot represented by $x$ for the program $y$. This is a primitive recursive function defined by cases:

$$SUCC(x, y) =$$

$$\begin{cases} 
\langle \ell(x) + 1, r(x) \rangle & \text{if } SKIP(x, y), \\
\langle \ell(x) + 1, r(x) \cdot p_{VAR}(\ell(x), y) \rangle & \text{if } INCR(x, y), \\
\langle \ell(x) + 1, \lfloor r(x)/p_{VAR}(\ell(x), y) \rfloor \rangle & \text{if } DECR(x, y), \\
\langle \min_{i \leq Lt(y + 1)}[\text{LABEL}(i, y) = \text{LABEL'}(\ell(x), y)] \rangle & \text{if } BRANCH(x, y), \\
\langle Lt(y + 1) + 1, r(x) \rangle & \text{otherwise.}
\end{cases}$$
The function

$$\text{INIT}^{(n)}(x_1, \ldots, x_n) = \langle 1, \prod_{i=1}^{n} (p_{2i})^{x_i} \rangle$$

gives the representation of the initial snapshot for inputs $x_1, \ldots, x_n$, and the predicate TERM given by

$$\text{TERM}(x, y) \iff \ell(x) > Lt(y + 1)$$

tests whether $x$ represents a terminal snapshot for program $y$. 
The function SNAP gives the numbers of successive snapshots produced by a program \( y \). This function is primitive recursive because

\[
\text{SNAP}^{(n)}(x_1, \ldots, x_n, y, 0) = \text{INIT}^{(n)}(x_1, \ldots, x_n)
\]

\[
\text{SNAP}^{(n)}(x_1, \ldots, x_n, y, i + 1) = \text{SUCC} \left( \text{SNAP}^{(n)}(x_1, \ldots, x_n, y, i), y \right).
\]

Thus,

\[
\text{STP}^{(n)}(x_1, \ldots, x_n, y, t) \iff \text{TERM} \left( \text{SNAP}^{(n)}(x_1, \ldots, x_n, y, t), y \right),
\]

hence \( \text{STP}^{(n)} \) is primitive recursive.
An important consequence is the next theorem known as the **Normal Form Theorem**:

**Theorem**

Let $f$ be a *partially computable function*. Then, there is a primitive recursive predicate $R(x_1, \ldots, x_n, y)$ such that

$$f(x_1, \ldots, x_n) = \ell \left( \min_z R(x_1, \ldots, x_n, z) \right).$$
Proof.

Let $y_0$ be the number of a program that computes $f(x_1, \ldots, x_n)$. Let $R(x_1, \ldots, x_n, z)$ be the predicate defined by

$$R(x_1, \ldots, x_n, z) \iff \text{STP}^{(n)}(x_1, \ldots, x_n, y_0, r(z)) \land (r(\text{SNAP}^{(n)}(x_1, \ldots, x_n, y_0, r(z))))_{1} = \ell(z).$$

Suppose that the right side of the above equality is defined. This means that there exists a number $z$ such that the computation of the program with number $y_0$ has reached a terminal snapshot in $r(z)$ or fewer steps and $\ell(z)$ is the value held in the output variable $Y$, that is, $\ell(z) = f(x_1, \ldots, x_n)$. If the right side is undefined it must be the case that $\text{STP}^{(n)}(x_1, \ldots, x_n, y_0, t)$ is false for all values of $t$, that is $f(x_1, \ldots, x_n) \uparrow$. 

\[ \square \]
A characterization of partially computable functions:

**Theorem**

A function is partially computable if and only if it can be obtained from the initial functions by a finite number of applications of composition, recursion, and minimalization.

**Proof.**

Every function that can be obtained by a finite number of applications of composition, recursion, and minimalization is clearly partially computable by previous results.
Proof cont’d

Proof.

Conversely, by the Normal Form Theorem, we can write any partially computable function as

\[ f(x_1, \ldots, x_n) = \ell \left( \min_z R(x_1, \ldots, x_n, z) \right), \]

where \( R \) is a primitive recursive predicate. Then \( R \) is obtained from initial functions by a finite number of applications of composition and recursion. Finally, the given function is obtained from \( R \) by one use of minimalization and then by application of \( \ell \).