Total and Partial Functions

• $f$ is a **total function** on the domain $Dom$ iff $f$ is defined on all elements of $Dom$.

• $f$ is a **partial function** on the domain $Dom$ iff $f$ is defined on zero or more elements of $Dom$. 
Total and Partial Functions

- The successor function $\text{succ}$:
  - Total function on $\mathbb{N}$.

- $\text{midchar}$, which returns the middle character of its argument string if there is one.
  - Partial function on the domain of strings.
  - Total function on:

- $\text{steps}$, defined on inputs of the form $<M, w>$, returns the number of steps that Turing machine $M$ executes, on input $w$, before it halts.
  - Partial function on the domain $\{<M, w>\}$.
  - Total function on:
Partially Computable Functions

Let $M = M = (K, \Sigma, \Gamma, \delta, s, \{h\})$.

The initial configuration of $M$ will be $(s, \square w)$.

Define $M(w) = z$ iff $(s, \square w) \vdash^*_{M} (h, \square z)$.

$M$ computes a function $f$ iff, for all $w \in \Sigma^*$:

- If $w$ is an input on which $f$ is defined,

  $M(w) = <f(w)>$

- Otherwise $M(w)$ does not halt.

A function $f$ is computable (or recursive) iff there exists a Turing machine $M$ that computes it and that always halts.
What if \( M \) Doesn’t Always Halt?

A function \( f \) is **partially computable** iff there exists a Turing machine \( M \) that computes it.

Let \( f \) be any partially computable function whose domain \( Dom \) is only a proper subset of \( \Sigma^* \).

- Suppose \( Dom \) is odd length strings.
- Suppose \( Dom \) is \( \{<M, w> : M \text{ halts on } w>\} \)
When $Dom$ is in D

Let $f$ be any partially computable function that is a total function of some decidable domain $Dom$.

Define $f'(z) = \begin{cases} f(z), & \text{if } z \in Dom \\ \text{Error}, & \text{else} \end{cases}$

Since $Dom$ is in D, there is some Turing machine $TF$ that decides it. If $M$ computes $f$ then $M'$ computes $f'$:

$M'(x) =$
1. Run $TF$ on $x$.
2. If it rejects, output $Error$.
3. If it accepts, run $M$ on $x$.

A function $f$ is **computable** iff there exists a Turing machine $M$ that computes either $f$ or $f'$ and that halts on all inputs.

Every computable function is also partially computable.
When *Dom* is Not a Set of Strings

*f* is computable iff all of the following conditions hold:

- There exist alphabets $\Sigma$ and $\Sigma'$.
- There exists an encoding of the elements of the domain of $f$ as strings in $\Sigma^*$.
- There exists an encoding of the elements of the range of $f$ as strings in $\Sigma''^*$.
- There exists some computable function $f'$ with the property that, for every $w \in \Sigma^*$:
  - If $w = <x>$ and $x$ is an element of $f'$'s domain, then $f'(w) = <f(x)>$, and
  - If $w$ is not the encoding of any element of $f'$'s domain (either because it is not syntactically well formed or because it encodes some value on which $f$ is undefined), then $f'(w) = Error$. 
A Computable Function

\[ \text{succ: } \mathbb{N} \rightarrow \mathbb{N}, \]
\[ \text{succ}(x) = x + 1 \]

We can encode the argument to \textit{succ} in unary and then define the following Turing machine \( M \) to compute it:

\[ M(x) = \]
1. Write 1.
2. Move left once.
3. Halt.
What About `midchar`?

`midchar` returns the middle character of its argument string if there is one.
What About \textit{midchar}?

\textit{midchar} returns the middle character of its argument string if there is one.

\[
M'(x) =
\begin{align*}
1. & \text{ If the length of } x \text{ is even, output } \textit{Error}. \\
2. & \text{ Otherwise find the middle character of } x \text{ and output it.}
\end{align*}
\]
What About steps?

steps(<M, w>) returns the number of steps that Turing machine M executes, on input w, before it halts.

The function steps is partially computable because the following three-tape Turing machine S computes it:

S(x) =
1. If x is not a syntactically well formed <M, w> string then loop.
2. If x is a syntactically well formed <M, w> string then:
   2.1 Copy M to tape 3.
   2.2 Copy w to tape 2.
   2.3 Write 0 on tape 1.
   2.4 Simulate M on w on tape 2, keeping a count on tape 1 of each step that M makes.
But steps is Not Computable

The proof is by reduction from $H$:

Suppose that there existed some Turing machine $ST$ that computed the function $steps'$, defined as:

$$steps'(<M, w>) =$$

if $M$ does not halt on $w$ then $Error$.
if $M$ halts on $w$ then the number of steps that $M$ executes before it halts.

Then the following Turing machine $DH$ would decide the language $H$:

$$DH(<M, w>) =$$

1. Run $ST(<M, w>)$.
2. If the result is $Error$ then reject. Else accept.

But we know that there can exist no Turing machine to decide $H$. So $ST$ must not exist.
There Exist Functions That Are Not Partially Computable

**Theorem:** There exist (a very large number of) functions that are not partially computable.

**Proof:** By a counting argument. Let $U$ be the set of unary functions from some subset of $\mathbb{N}$ to $\mathbb{N}$. Encode both inputs and outputs as binary strings.
There Exist Functions That Are Not Partially Computable

**Lemma:** There is a countably infinite number of partially computable functions in $U$.

**Proof of Upper bound:** Every partially computable function in $U$ is computed by some Turing machine $M$ with $\Sigma$ and $\Gamma$ equal to $\{0, 1\}$, There are countably infinitely many such machines. There cannot be more partially computable functions than there are Turing machines.

**Proof of Lower bound:** The number of partially computable functions must be infinite because it includes all the constant functions:

$$cf_1(x) = 1, \; cf_2(x) = 2, \; cf_3(x) = 3, \ldots$$

So there is a countably infinite number of partially computable functions in $U$. 
There Exist Functions That Are Not Partially Computable

**Lemma:** There is an uncountably infinite number of functions in $U$.

**Proof of Lemma:** Let $S$ be $P(\mathbb{N})$. For any element $s$ of $S$, let $f_s$ be the characteristic function of $s$. No two elements of $S$ have the same characteristic function. There is an uncountably infinite number of elements in $S$, so there is an uncountably infinite number of such characteristic functions, each of which is in $U$. 
There Exist Functions That Are Not Partially Computable

*Proof of Theorem:* Since there is only a countably infinite number of partially computable functions in $U$ and an uncountably infinite number of functions in $U$, there is an uncountably infinite number of functions in $U$ that are not partially computable.
A Function that Isn’t Partially Computable

Let $E$ be a lexicographic enumeration of the TMs that compute the partially computable functions in $U$.

Let $M_i$ be the $i$th machine in that enumeration.

Define a new function $notcomp(x)$ as follows:

$$notcomp: \mathbb{N} \to \{0, 1\},$$

$$notcomp(x) = 1 \text{ if } M_{x}(x) = 0,$$

$$0 \text{ otherwise}.$$
A Function that Isn’t Partially Computable

\[ \text{notcomp}(x) = 1 \text{ if } M_x(x) = 0, \]
\[ 0 \text{ otherwise.} \]

So \( \text{notcomp}(x) = 0 \) if either:

- \( M_i(x) \) is defined and the value is something other than 0, or
- \( M_i(x) \) is not defined.

This new function \( \text{notcomp} \) is in \( U \), but it differs, in at least one place, from every function that is computed by a Turing machine whose encoding is listed in \( E \). So there is no Turing machine that computes it. Thus it is not partially computable.
Let $T$ be the set of all standard TMs that:

- Have tape alphabet $\Gamma = \{\square, 1\}$, and
- Halt on a blank tape.

Define the **busy beaver functions** $S(n)$ and $\Sigma(n)$:

- $S(n)$: the maximum number of steps that are executed by any element of $T$ with $n$-nonhalting states, when started on a blank tape, before it halts.

- $\Sigma(n)$: the maximum number of $1$’s that are left on the tape by any element of $T$ with $n$-nonhalting states, when it halts.
The Busy Beaver Functions

- $S(n)$: the maximum number of steps.
- $\Sigma(n)$: the maximum number of 1’s.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$S(n)$</th>
<th>$\Sigma(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>21</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>107</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>$\geq 47,176,870$</td>
<td>4098</td>
</tr>
<tr>
<td>6</td>
<td>$\geq 3\times10^{1730}$</td>
<td>$\geq 1.29\times10^{865}$</td>
</tr>
</tbody>
</table>
The Busy Beaver Functions are Total

**Theorem:** Both $S$ and $\Sigma$ are total functions on the positive integers.

**Proof:** For any value $n$, both $S(n)$ and $\Sigma(n)$ are defined iff there exists some standard Turing machine $M$, with tape alphabet $\Gamma = \{\Box, 1\}$, where:

- $M$ has $n$ nonhalting states, and
- $M$ halts on a blank tape.

Name the nonhalting states of $M$ with the integers 1, …, $n$. Then $M$ is:
Monotonicity

**Theorem:** Both $S$ and $\Sigma$ are strictly monotonically increasing functions. In other words:

- $S(n) < S(m)$ iff $n < m$, and
- $\Sigma(n) < \Sigma(m)$ iff $n < m$.

**Proof:** We prove four claims:

- $n < m \rightarrow S(n) < S(m)$.
- $S(n) < S(m) \rightarrow n < m$.
- $n < m \rightarrow \Sigma(n) < \Sigma(m)$.
- $\Sigma(n) < \Sigma(m) \rightarrow n < m$. 
Theorem: Neither $S$ nor $\Sigma$ is computable.

Proof that $S$ is not computable: If it were, there would be some TM $BB$, with $b$ states, that computes it. Define:

- For any positive integer $n$, a Turing machine $Write_n$ that writes $n$ $1$’s on its tape, one at a time, moving rightwards, and then halts with its read/write head on the blank square immediately to the right of the rightmost $1$. $Write_n$ has $n$ nonhalting states plus one halting state.

- $Multiply$, which multiplies two unary numbers, written on its tape and separated by the character #. Let $m$ be the number of states in $Multiply$. 
Define $\text{Trouble}_n$:

$$\text{Write}_n ; R \text{ Write}_n L \text{ Multiply}_n L \text{ BB}$$

$\text{Trouble}_n$:

- Writes a string of the form $1^n;1^n$.
- Moves its read/write head back to the left
- Invokes $\text{Multiply}$, which results in the tape containing a string of exactly $n^2$ $1$’s.
- It moves its read/write head back to the left.
- Invokes $\text{BB}$, which outputs $S(n^2)$. 

Computability
The Number of States in $\text{Trouble}_n$

$> \text{Write}_n \ ; \ R \ \text{Write}_n \ \text{L} \ \text{Multiply} \ \text{L} \ \text{BB}$

<table>
<thead>
<tr>
<th>Component</th>
<th>Number of States</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Write}_n$</td>
<td>$n + 1$</td>
</tr>
<tr>
<td>; R</td>
<td>1</td>
</tr>
<tr>
<td>$\text{Write}_n$</td>
<td>$n + 1$</td>
</tr>
<tr>
<td>$\text{L}$</td>
<td>2</td>
</tr>
<tr>
<td>$\text{Multiply}$</td>
<td>$m$</td>
</tr>
<tr>
<td>$\text{L}$</td>
<td>2</td>
</tr>
<tr>
<td>$\text{BB}$</td>
<td>$b$</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>$2n + m + b + 7$</td>
</tr>
</tbody>
</table>
The Bottom Line

Since $BB$ writes a string of length $S(n^2)$ and it can write only one character per step, $Trouble_n$ must run for at least $S(n^2)$ steps. If $n > 0$, $Trouble_n$ is a TM with $2n + m + b + 7$ states that runs for at least $S(n^2)$ steps. So:

$$S(2n + m + b + 7) \geq S(n^2)$$

$S$ is monotonically increasing, so it must also be true that, for any $n > 0$:

$$2n + m + b + 8 \geq n^2$$

But, since $n^2$ grows faster than $n$ does, that cannot be true.

In assuming that $BB$ exists, we have derived a contradiction. So $BB$ does not exist. So $S$ is not computable.
# Three Views

<table>
<thead>
<tr>
<th>The Problem View</th>
<th>The Language View</th>
<th>The Functional View</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given three natural numbers, $x$, $y$, and $z$, is $z = x \cdot y$?</td>
<td>${ x \cdot y = z : x, y, z \in {0, 1}^*, num(x) \cdot num(y) = num(z) }$</td>
<td>$f: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, $f(x, y) = x \cdot y$ computable</td>
</tr>
<tr>
<td>Given a TM $M$, does $M$ have an even number of states?</td>
<td>${ &lt;M&gt; : M \text{ has an even number of states} }$</td>
<td>$f: {&lt;M&gt;} \rightarrow \text{Boolean}$, $f(&lt;M&gt;) = \text{True}$ if $M$ has an even number of states, $\text{False}$ otherwise computable</td>
</tr>
<tr>
<td>Given a TM $M$ and a string $w$, does $M$ halt on $w$ in $n$ steps?</td>
<td>${ &lt;M, w, n&gt; : M \text{ halts on } w \text{ in } n \text{ steps} }$</td>
<td>$f: {&lt;M, w&gt;} \rightarrow \mathbb{N}$, $f(&lt;M, w&gt;) = \text{If } TM \ M \text{ halts on } w, \text{ then the number of steps it executes before halting, else undefined.}$ partially computable</td>
</tr>
<tr>
<td>Given a TM $M$, does $M$ halt on all strings in no more than $n$ steps?</td>
<td>${ &lt;M, n&gt; : M \text{ halts on each element of } \Sigma^* \text{ in no more than } n \text{ steps} }$</td>
<td>$f: {&lt;M&gt;} \rightarrow \mathbb{N}$, $f(&lt;M&gt;) = \text{If } TM \ M \text{ halts on all strings, then the maximum number of steps it executes before halting, else undefined.}$ not partially computable</td>
</tr>
</tbody>
</table>
Recursive Function Theory

A function is computable iff there is a Turing machine that computes it and always halts.

Is there an alternative, nonprocedural definition?

Yes.

For the rest of this discussion, define:

- A **recursive function** is a computable function.
- A **partial recursive function** is a partially computable function.

Why use *recursive* as a synonym for *computable*?
The set of primitive recursive functions is the smallest set that includes:
- The constant function 0.
- $\text{succ}(n) = n + 1$.
- A family of projection functions: $(n_1, n_2, \ldots n_k) = n_j$.

and is closed under:
- Composition of $g$ with $h_1, h_2, \ldots h_k$:
  $$g(h_1(\_), h_2(\_), \ldots h_k(\_))$$
- Primitive recursion of $f$ in terms of $g$ and $h$:
  $$f(n_1, n_2, \ldots n_k, 0) = g(n_1, n_2, \ldots n_k)$$
  $$f(n_1, n_2, \ldots n_k, m+1) = h(n_1, n_2, \ldots n_k, m, f(n_1, n_2, \ldots n_k, m))$$
Examples

\[\text{plus}(n, 0) = p_{1,1}(n) = n.\]
\[\text{plus}(n, m+1) = \text{succ}(p_{3,3}(n, m, \text{plus}(n, m))).\]

For clarity, we will simplify our future definitions by omitting the explicit calls to the projection functions. Doing that here, we get:

\[\text{plus}(n, 0) = n.\]
\[\text{plus}(n, m+1) = \text{succ}(\text{plus}(n, m)).\]
Examples

\(\text{times}(n, 0) = 0.\)
\(\text{times}(n, m+1) = \text{plus}(n, \text{times}(n, m)).\)

\(\text{factorial}(0) = 1.\)
\(\text{factorial}(n + 1) = \text{times}(\text{succ}(n), \text{factorial}(n)).\)

\(\exp(n, 0) = 1.\)
\(\exp(n, m+1) = \text{times}(n, \exp(n, m)).\)

\(\text{pred}(0) = 0.\)
\(\text{pred}(n+1) = n.\)
Theorem: Every primitive recursive function is computable.

Proof: Each of the basic functions, as well as the two combining operations can be implemented in a straightforward fashion on a Turing machine or using a standard programming language.
Theorem: Not all computable functions are primitive recursive.

Proof: Lexicographically enumerate the unary primitive recursive functions, $f_0$, $f_1$, $f_2$, $f_3$, ....

Define $g(n) = f_n(n) + 1$. $g$ is computable, but it is not on the list. Suppose it were $f_m$ for some $m$. Then $f_m(m) = f_m(m) + 1$, which is absurd.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_0$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f_3$</td>
<td></td>
<td></td>
<td></td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>$f_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Suppose $g$ were $f_3$. Then $g(3) = 27 + 1$. 
A Function that Isn’t Primitive Recursive

Ackermann’s function:

\[ A(0, y) = y + 1. \]
\[ A(x + 1, 0) = A(x, 1). \]
\[ A(x + 1, y + 1) = A(x, A(x + 1, y)). \]
# Ackermann’s Function

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>13</td>
<td>29</td>
<td>61</td>
<td>125</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td>65533</td>
<td>$2^{65536}-3$</td>
<td>*</td>
<td>$2^{2^{65536}}-3$ #</td>
</tr>
</tbody>
</table>

* 19,729 digits  
# 10^{5940} digits

3·10^{17} seconds since Big Bang  
10^{79} atoms in observable universe

Thus writing digits at the speed of light on all atoms in the universe starting at the Big Bang would have produced 3·10^{96} digits.
Minimalization

The \textit{minimalization} \( f \) of a function \( g \) (of \( k + 1 \) arguments) is a function of \( k \) arguments:

\[
f(n_1, n_2, \ldots n_k) = \text{the smallest } m \text{ such that:} \\
g(n_1, n_2, \ldots n_k, m) = 1, \text{ if there is such an } m, \\
0, \text{ otherwise.}
\]

Given any function \( g \) and any set of \( k \) arguments to it, there either is at least one value \( m \) such that
\[
g(n_1, n_2, \ldots n_k, m) = 1 \text{ or there isn’t.}
\]

If there is at least one such value, then there is a smallest one. So there always exists a function \( f \) that is the minimalization of \( g \).
Minimalization

If $g$ is computable, then we can build a Turing machine $T_{\text{min}}$ that almost computes $f$ as follows:

$$T_{\text{min}}(n_1, n_2, \ldots n_k) =$$

1. $m = 0$.
2. While $g(n_1, n_2, \ldots n_k, m) \neq 1$ do:
   - $m = m + 1$.
3. Return $m$. 
The Problem with $T_{\text{min}}$

\[ T_{\text{min}}(n_1, n_2, \ldots n_k) = \]
\begin{enumerate}
\item $m = 0.$
\item While $g(n_1, n_2, \ldots n_k, m) \neq 1$ do:
    \begin{itemize}
    \item $m = m + 1.$
    \end{itemize}
\item Return $m.$
\end{enumerate}

$T_{\text{min}}$ will not halt if no value of $m$ exists. There is no way for $T_{\text{min}}$ to discover that no such value exists and thus return 0.

A function $g$ is \textit{minimalizable} iff, for every $n_1, n_2, \ldots n_k$, there is an $m$ such that $g(n_1, n_2, \ldots n_k, m) = 1.$
A function is $\mu$-recursive iff it can be obtained from the basic functions:

- The constant function 0.
- $\text{succ}(n) = n + 1$.
- A family of projection functions: $(n_1, n_2, \ldots, n_k) = n_j$.

Using the operations of:

- Composition,
- Recursive definition, and
- Minimalization of minimalizable functions.
Primitive Recursive vs \( \mu \)-Recursive Functions

- Primitive recursive: Use iteration.
- \( \mu \)-recursive: Use while loop.
Equivalence of $\mu$-Recursion and Computability

**Theorem:** A function is $\mu$-recursive iff it is computable.

**Proof:** By two constructions that show that each can simulate the other:

- Every partial $\mu$-recursive function is partially computable. We show this by showing how to build a Turing machine for each of the basic functions and for each of the combining operations.

- Every partially computable function is partial $\mu$-recursive. We show this by showing how to construct $\mu$-recursive functions to perform each of the operations that a Turing machine can perform.
Partial $\mu$-Recursive Functions

A function is **partial $\mu$-recursive** iff it can be obtained from the basic functions:

- the constant function 0,
- the successor function: $\text{succ}(n) = n + 1$,
- the family of projection functions: for any $k \geq j > 0$,
  $$p_{k,j}(n_1, n_2, \ldots n_k) = n_j,$$

and that is closed under the operations:

- composition of $g$ with $h_1, h_2, \ldots h_k$,
- primitive recursion of $f$ in terms of $g$ and $h$, and
- minimalization (of any, possibly nonminimalizable function).
Equivalence of Partial $\mu$-Recursion and Partial Computability

**Theorem**: A function is partial $\mu$-recursive iff it is partially computable.

**Proof**: By two constructions that show that each can simulate the other.
Functions and Machines

Partial Recursive Functions

Recursive Functions

Primitive Recursive Functions

Turing Machines
A Virus Program

\[ \text{virus()} = \]
1. For each address in address book do:
   1.1. Write a copy of myself.
   1.2. Mail it to the address.
2. Do something fun and malicious like change one bit in every file on the machine.
3. Halt.

Can we implement step 1.1?

It won’t do to write something like:

Write “Write” || “a_1 \ a_2 \ a_3 \ldots \ a_n”
Isolating the Problem

\textit{virus() =}
\begin{enumerate}
\item \textit{copyme} = copy of myself.
\item For each address in address book do:
  \begin{enumerate}
  \item Mail \textit{copyme} to the address.
  \item Do something fun and malicious like change one bit in every file on the machine.
  \item Halt.
  \end{enumerate}
\end{enumerate}

Now we just need a way for \textit{virus} to get one copy of itself onto its tape.

Describe virus as having two parts:
\begin{enumerate}
\item Get its description onto its tape.
\item Work.
\end{enumerate}

Further decomposition:
\begin{enumerate}
\item \textit{A}; \textit{B}.
\item Work.
\end{enumerate}
Printing Functions

For any literal string $s$, $P_s$ is the description of a Turing machine that writes the string $s$ onto the tape; $s$ is hardwired into $P_s$.

Example: $P_{abbb} = <aRbRbRbR>$. 

Notice that the length of $P_s$ depends on the length of $s$. 
Printing Functions

Define a Turing machine, $createP$, that takes a string $s$ as input on one tape and outputs the printing function $P_s$ on a second tape:

$createP(s) =$
1. For each character $c$ in $s$ (on tape 1) do on tape 2:
   1.1. Write $c$.
   1.2. Write R.

Notice that the length of $createP$ is fixed.
A and B

A will write \(<B, work>\), onto the tape.

The string \(<B, work>\) will be hardwired into A, so the length of A itself depends on \(|<B, work>|\). When A is done, the tape will be:

\[
\begin{array}{c}
<B> \quad <work>
\end{array}
\]

B will write \(<A>\), the description of A, onto the tape immediately to the left of what A wrote. So, after B has finished, the job of copying virus will be complete and the tape will be:

\[
\begin{array}{c}
<A> \quad <B> \quad <work>
\end{array}
\]

The length of B cannot depend on \(<A>\). So how does it work?
1. /* Invoke createP to write onto tape 2 the code that writes the string that is currently on tape 1. For each character c in s (on tape 1) do on tape 2:
   Write c.
   Write R.
2. Starting at the rightmost character c on tape 2 and the blank immediately to the left of the leftmost character on tape 1, loop until all characters have been processed:
   Copy c to tape 1.
   Move both read/write heads one square to the left.

So the code for B (unlike the code for A) is independent of the particular Turing machine of which we need to make a copy.
The Operation of B

Before B starts

After step 1

After step 2
Virus

virus() =
  A: P_{<B><work>}.  
  B: createP.  
  work.
ObtainSelf

Any program $M$ that uses its own description:

1. $obtainSelf$.
2. $work$ (which may exploit the description that $obtainSelf$ produced).

The definition of $obtainSelf$, which constructs $<M>$:

$$obtainSelf(work) = P_{<B><work>}$.
$$B: createP.$$
The Recursion Theorem

**Theorem:** For every Turing machine $T$ that computes a partially computable function $t$ of two string arguments, there exists a Turing machine $R$ that computes a partially computable function $r$ of one string argument and:

$$\forall x \ (r(x) = t(<R>, x)).$$

**Proof:** The proof is by construction. The construction is identical to the one we showed above in our description of *virus* except that we substitute $T$ for *work*.
Gödel Numbering

A one-to-one function that assigns natural numbers to objects is called a **Gödel numbering**.

**Gödel numbering of Turing machines:** Let $M$ be a Turing machine that computes some partially computable function. Let $\langle M \rangle$ be the string description of $M$. Our encoding scheme uses nine symbols, which can be encoded in binary using four bits. Rewrite $\langle M \rangle$ as a binary string. Now view that string as the number it encodes.

**Gödel numbering of partial recursive functions:** assign to each function the smallest number that has been assigned to some Turing machine that computes it.

Define:

$$\varphi_k$$ to be the partially computable function with Gödel number $k$. 
Theorem: For all $m, n \geq 1$, there exists a computable function $s_{m,n}$ such that, if $k$ is the Gödel number of some partially computable function of $m+n$ arguments, then:

For all $k, v_1, v_2, \ldots, v_m, y_1, y_2, \ldots, y_n$:

- $s_{m,n}(k, v_1, v_2, \ldots, v_m)$ returns a number $j$ that is the Gödel number of some partially computable function of $n$ arguments, and

- $\varphi_j(y_1, y_2, \ldots, y_n) = \varphi_k(v_1, v_2, \ldots, v_m, y_1, y_2, \ldots, y_n)$.

Proof: $M_{m,n}$ computes $s_{m,n}$. On input $(k, v_1, v_2, \ldots, v_m)$, $M_{m,n}$ builds $M_j$ that operates as follows on input $w$: Write $v_1, v_2, \ldots, v_m$ on the tape immediately to the left of $w$; move the read/write head all the way to the left in front of $v_1$; and pass control to the Turing machine encoded by $k$. $M_{m,n}$ will then return $j$, the Gödel number of the function computed by $M_j$. 

\[ w \]