The next thing we want to prove is the following:

**Theorem 6.1:** If there is a Post-Turing program that computes the partial function $f(x_1, \ldots, x_m)$, then $f$ is partially computable.

Since our definition of partial computability is based on the language $L$, this theorem states the following:

If the $m$-ary partial function $f$ on $A^*$ is computed by a program of $T$, then there is a program of $L$ that computes $f$ (using base $n$ values of strings).

Based on Theorem 6.1 and also Theorems 3.2 and 5.1, we can derive another Theorem:

**Theorem 6.2:** Let $f$ be an $m$-ary partial function on $A^*$, where $A$ is an alphabet of $n$ symbols. Then the following conditions are all equivalent:
1. $f$ is partially computable;
2. $f$ is partially computable in $L_n$;
3. $f$ is computed strictly by a Post-Turing program;
4. $f$ is computed by a Post-Turing program.

The fact that there are so many equivalent notions of computability constitutes important evidence for the correctness of Church’s Thesis.

For the proof of Theorem 6.1, let us now consider an $m$-ary partial function on $N$.

**Corollary 6.3:** For any $n, l \geq 1$, an $m$-ary partial function $f$ on $N$ is partially computable in $L_l$ if and only if it is also partially computable in $L_n$.

**Proof:** Each of these conditions is equivalent to the function $f$ being partially computable.

Considering the language $L_1$, we have:

**Corollary 6.4:** Every partially computable function is computed strictly by some Post-Turing program that uses only the symbols $s_0$ and $s_1$.

**Proof:** This follows immediately from the fact that we can simulate $L_n$ in $T$, as shown before.

Now let $\varphi$ be a Post-Turing program that computes $f$. We want to construct a program $Q$ in the language $L$ that computes $f$.

$Q$ will consist of three sections:

**BEGINNING**

**MIDDLE**

**END**

The MIDDLE section will simulate $\varphi$ in a step-by-step “interpretive” manner.

The BEGINNING section will arrange the input to $Q$ in the appropriate format for MIDDLE.

The END section will extract the output.

Let $f$ be an $m$-ary partial function on $A^*$, where $A = \{s_1, \ldots, s_r\}$. The Post-Turing program $\varphi$ will also use the blank $B$ and perhaps other symbols $s_{r+1}, \ldots, s_r$ (we are not assuming that the computation is strict).

We write the symbols that $\varphi$ uses in the following order:

$s_1, \ldots, s_n, s_{n+1}, \ldots, s_r, B$.

The program $Q$ will simulate $\varphi$ by using the numbers that strings on this alphabet represent in base $r + 1$ as "codes" for the corresponding strings.
Simulation of $T$ in $L$

Note that we will write the blank as $s_{r+1}$ instead of $s_0$. The current tape configuration will be kept track of by $Q$ using three numbers stored in the variables $L$, $H$, and $R$.

$H$ will contain the numerical value of the symbol currently being scanned by the head.

$L$ will contain a number that represents in base $r + 1$ a string of symbols $w$ such that the tape contents to the left of the head consists of infinitely many blanks followed by $w$.

$R$ will contain a number that represents in a similar manner the string of symbols to the right of the string.

Example:

Consider the following tape configuration for $r = 3$ (so we will use base 4):

$$\cdots B B B s_1 s_3 s_2 B s_1 s_2 B B \cdots$$

Obviously, $H = 2$.

$L = 1 \cdot 4 + 3 = 7$

$R = 4 \cdot 4^2 + 1 \cdot 4 + 2 = 70$ (remember that $B = s_{r+1}$)

Simulation of $T$ in $L$

We are now able to simulate all of the instruction types of $T$ by programs of $L$.

As usual, we will specify how to simulate each type of instruction.

An instruction PRINT $s_i$ is simulated in $L$ by

$$H \leftarrow i$$

An instruction IF $s_i$ GOTO L is simulated in $L$ by

$$IF \ H = i \ GOTO \ L$$

Simulation of $T$ in $L$

An instruction RIGHT is simulated in $L$ by

$$L \leftarrow \text{CONCAT}_{r+1}(L, H)$$

$$H \leftarrow \text{RTEND}_{r+1}(R)$$

$$R \leftarrow \text{LTRUNC}_{r+1}(R)$$

$$IF \ R \neq 0 \ GOTO \ E$$

$$R \leftarrow r + 1$$

// if $R = 0$, then there is a blank

// ($s_{r+1}$) to the right of the head

Simulation of $T$ in $L$

An instruction LEFT is simulated in $L$ by

$$R \leftarrow \text{CONCAT}_{r+1}(H, R)$$

$$H \leftarrow \text{RTEND}_{r+1}(L)$$

$$L \leftarrow \text{LTRUNC}_{r+1}(L)$$

$$IF \ L \neq 0 \ GOTO \ E$$

$$L \leftarrow r + 1$$

// if $L = 0$, then there is a blank

// ($s_{r+1}$) to the left of the head

Simulation of $T$ in $L$

Now the MIDDLE section of $Q$ can be generated by replacing each instruction in $\varphi$ by its simulation.

When we write BEGINNING and END, we must consider the following:

$f$ is an $m$-ary function on $\{s_1, \ldots, s_n\}^*$.

Thus the initial values of $X_1, \ldots, X_m$ for $Q$ are numbers that represent the input strings in base $n$.

However, the computations in $Q$ assume a base $r + 1$ representation of the input strings.
Simulation of $T$ in $L$

Fortunately, we have Theorem 1.1 of Chapter 5, which tells us that the functions $\text{UPCHANGE}_{n,r}$ and $\text{DOWNCHANGE}_{n,r}$ are computable.

Now we can develop the BEGINNING section. Its task is to calculate the initial values of $L$, $H$, and $R$, which represent the initial tape configuration

$$B \ x_1 \ B \ x_2 \ B \ ... \ B \ x_m,$$

where the numbers $x_1, ..., x_m$ are represented in base $n$ notation.

The BEGINNING section then looks like this:

$$L \leftarrow r + 1$$
$$H \leftarrow r + 1$$
$$Z_1 \leftarrow \text{UPCHANGE}_{n,r+1}(X_1)$$
$$Z_2 \leftarrow \text{UPCHANGE}_{n,r+1}(X_2)$$
$$\vdots$$
$$Z_m \leftarrow \text{UPCHANGE}_{n,r+1}(X_m)$$
$$R \leftarrow \text{CONCAT}_{r+1}(Z_1, r + 1, Z_2, r + 1, ..., r + 1, Z_m)$$

Now let us write the END section. It is supposed to put the output of the simulated program in base $n$ representation into variable $Y$. Since we are not demanding a strict computation by $\psi$, the output is the concatenation of all symbols on the tape that belong to the alphabet $A = \{s_1, ..., s_n\}$.

Therefore, our END section is written as follows:

$$Z \leftarrow \text{CONCAT}_{r+1}(L, H, R)$$
$$Y \leftarrow \text{DOWNCHANGE}_{n,r+1}(Z)$$

(Remember that $\text{DOWNCHANGE}_{n,r+1}$ automatically removes all symbols $s_{r+1}, ..., s_n$).

Turing Machines

Actually, Turing’s original model of a computer was different from the Post-Turing language. We will now look at a model that is closer to Turing’s original idea, the Turing machine. Its main difference to the Post-Turing language is that it does not read a list of instructions, but it is defined by a finite number of internal states and transitions between them.

The machine’s next state is always determined by its current state and the current symbol under the head. Each transition also includes printing a symbol or moving the head one square to the left or right.

The states are represented by the symbols $q_1, q_2, q_3, ..., s_p, s_1, s_2, ...,$ where as usual $s_0 = B$ is the blank.

Expressions of one of the following three forms will be referred to as quadruples:

1. $q_i \ s_j \ s_k \ q_l$
2. $q_i \ s_j \ R \ q_l$
3. $q_i \ s_j \ L \ q_l$

They indicate that if when we are in state $q_i$ and the current symbol under the head is $s_j$, the machine goes into state $q_l$ and

1. prints symbol $s_k$ at the current head position,
2. moves the head one square to the right, or
3. moves the head one square to the left.

We now define a Turing machine to be a finite set of quadruples, no two of which begin with the same pair $q_i, s_j$.

This means that at any time during the computation there is no ambiguity about the next action to be performed.

A Turing Machine without this requirement is called a nondeterministic Turing machine.

In this course, however, we will only consider deterministic Turing machines.
Turing Machines

The **alphabet** of a given Turing machine \( \mathcal{M} \) consists of all the symbols \( s_i \) which occur in quadruples of \( \mathcal{M} \) except the symbol \( s_0 \).

A Turing machine **begins** its computation in state \( q_1 \).

A Turing machine **halts** if it is in state \( q_i \), scans the symbol \( s_j \), and does not have a quadruple that begins with \( q_i \ s_j \).

We use the same conventions with regard to **input** and **output** as we did for Post-Turing programs.

So it should be clear what it means to say that some given Turing machine \( \mathcal{M} \) computes a partial function \( f \) on \( A^* \) for a given alphabet \( A \).

**Example:** Consider the following Turing machine with alphabet \( A = \{1\} \) (we will write \( s_0 = B \) and \( s_1 = 1 \)):

\[
\begin{array}{ccc}
 q_1 & B & R \ q_2 \\
 q_2 & 1 & R \ q_2 \\
 q_2 & B & 1 \ q_3 \\
 q_3 & 1 & R \ q_3 \\
 q_3 & B & 1 \ q_1 \\
\end{array}
\]

... or a table:

<table>
<thead>
<tr>
<th>B</th>
<th>( q_1 )</th>
<th>( q_2 )</th>
<th>( q_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( R \ q_2 )</td>
<td>( 1 \ q_3 )</td>
<td>( 1 \ q_1 )</td>
</tr>
</tbody>
</table>

We can also represent the machine by a state transition diagram...

![State Transition Diagram]

(We indicate the current state of a Turing machine by writing the state symbol below the arrow.)

Turing Machines

Just as for Post-Turing programs, we may speak of a Turing machine \( \mathcal{M} \) to compute a function **strictly**.

If \( \mathcal{M} \) computes \( f \), where \( f \) is a partial function on \( A^* \), we say that \( \mathcal{M} \) computes \( f \) strictly if

1. the alphabet of \( \mathcal{M} \) is a subset of \( A \) and
2. whenever \( \mathcal{M} \) halts, the final configuration has the form:

\[
\begin{array}{c}
\uparrow \\
q_i
\end{array}
\]

(We indicate the current state of a Turing machine by writing the state symbol below the arrow.)