Numerical Representation of Strings
This program computes \( \text{DOWNCHANGEN}_n \):

\[
\text{[A]} \quad \begin{align*}
\text{IF } X &= 0 \text{ GOTO E} \\
Z &\gets \text{LTEND}(X) \quad \text{ // Z receives leftmost symbol} \\
X &\gets \text{LTRUNC}(X) \quad \text{ // removes this symbol from x} \\
\text{IF } Z > n &\text{ GOTO A} \quad \text{ // check if we must cross out Z} \\
Y &\gets n \cdot Y + Z \quad \text{ // if not, add digit Z to output} \\
\text{GOTO } A
\end{align*}
\]

A Progr. Language for String Computations
The instructions in our programming language \( \mathcal{L} \) do not seem well-designed for string computations. Let us consider the instruction type \( V \leftarrow V + 1 \).

For example, if we use the alphabet \{a, b, c, d\}, and the variable \( V \) contains the number corresponding to the string \( \text{cadd} \), what will this instruction do? It will turn \( \text{cadd} \) into \( \text{cbaa} \).

Why would we want to have this as one of our fundamental operations in the programming language?

A Progr. Language for String Computations
To improve this situation, we will now introduce specific programming languages \( \mathcal{L}_n \) for computations on alphabets of size \( n \) for all \( n > 0 \). All variables will be the same as in \( \mathcal{L} \), except that we now consider their values to be from \( A^* \) with \( |A| = n \).

The use of labels, formal rules of syntax, and macro expansions will also be identical to \( \mathcal{L} \).

An m-ary partial function on \( A^* \) which is computed by a program in \( \mathcal{L}_n \) is said to be partially computable in \( \mathcal{L}_n \). If the function is total and partially computable in \( \mathcal{L}_n \), it is called computable in \( \mathcal{L}_n \).

A Progr. Language for String Computations
Although the instructions of \( \mathcal{L}_n \) refer to strings, we can still think of them as referring to the numbers associated with the strings.

For example, given an alphabet with \( n \) symbols, the instruction

\[
V \leftarrow sV
\]

has the effect of replacing the current value of the variable \( V \) with the value

\[
i \cdot n^{|V|} + V.
\]

So you see that the “natural” string operations in \( \mathcal{L}_n \) have complex numerical counterparts.

A Progr. Language for String Computations
Now let us look at some useful macros for use in \( \mathcal{L}_n \) and their expansions:

1. The macro IF \( V=0 \) GOTO L has the expansion

\[
\text{IF } V \text{ ENDS } s_1 \text{ GOTO } L \\
\text{IF } V \text{ ENDS } s_2 \text{ GOTO } L \\
\vdots \\
\text{IF } V \text{ ENDS } s_n \text{ GOTO } L
\]
2. The macro \( V \leftarrow 0 \) has the expansion

\[
[A] \quad V \leftarrow V \\
\quad \text{IF } V \neq 0 \text{ GOTO } A
\]

3. The macro \( \text{GOTO } L \) has the expansion

\[
Z \leftarrow 0 \\
Z \leftarrow s_iZ \\
\text{IF } Z \text{ ENDS } s_i \text{ GOTO } L
\]

4. The macro \( V' \leftarrow V \) has a complicated expansion. So let us first introduce the description

\[
\text{IF } V \text{ ENDS } s_i \text{ GOTO } B_i \quad (1 \leq i \leq n)
\]

to stand for

\[
\text{IF } V \text{ ENDS } s_1 \text{ GOTO } B_1 \\
\text{IF } V \text{ ENDS } s_2 \text{ GOTO } B_2 \\
\vdots \\
\text{IF } V \text{ ENDS } s_n \text{ GOTO } B_n
\]

This is also called a filter.

Finally, let us look at two useful functions, namely \( f(x) = x + 1 \) and \( g(x) = x - 1 \). We want to show that these functions are computable in \( \mathcal{L}_n \) by writing programs that compute them.

### Example 1: An \( \mathcal{L}_n \) program that computes \( f(x) = x + 1 \)

\[
[B] \quad \text{IF } X \text{ ENDS } s_i \text{ GOTO } A_i \quad (1 \leq i \leq n) \\
\quad Y \leftarrow s_iY \\
\quad \text{GOTO } E
\]

\[
[A] \quad X \leftarrow X \\
\quad (1 \leq i \leq n) \\
\quad Y \leftarrow s_{i+1}Y \\
\quad \text{GOTO } C : \\
\]

\[
[A_1] \quad X \leftarrow X \\
\quad (1 \leq i \leq n) \\
\quad Y \leftarrow s_iY \\
\quad \text{GOTO } B \\
\]

### Example 2: An \( \mathcal{L}_n \) program that computes \( g(x) = x - 1 \)

\[
[B] \quad \text{IF } X \text{ ENDS } s_i \text{ GOTO } A_i \quad (1 \leq i \leq n) \\
\quad \text{GOTO } E \\
\]

\[
[A] \quad X \leftarrow X \\
\quad (1 < i \leq n) \\
\quad Y \leftarrow s_iY \\
\quad \text{GOTO } C : \\
\]

\[
[A_2] \quad X \leftarrow X \\
\quad \text{IF } X \neq 0 \text{ GOTO } C_2 \\
\quad \text{GOTO } E \\
\]

\[
[C_2] \quad Y \leftarrow s_iY \\
\quad \text{GOTO } B \\
\]

\[
[C] \quad \text{IF } X \text{ ENDS } s_i \text{ GOTO } D_i \quad (1 \leq i \leq n) \\
\quad \text{GOTO } E \\
\]

\[
[D] \quad X \leftarrow X \\
\quad (1 \leq i \leq n) \\
\quad Y \leftarrow s_iY \\
\quad \text{GOTO } C : \\
\]
The Languages $L$ and $L_n$

Now that you know the language $L$ and the different languages $L_n$, do you think that they are equivalent, i.e. their programs can compute the same functions?

Well, you probably have the feeling that they are. And they are indeed.

This means that a function $f$ is partially computable if and only if it is partially computable in each $L_n$. Therefore, $f$ is partially computable in any one $L_n$ if and only if it is partially computable in all of them.

Theorem 3.1: A function is partially computable if and only if it is partially computable in $L_1$.

Proof: It is easy to see that the languages $L$ and $L_1$ are really the same. That is, the numerical effect of the instructions $V \leftarrow s_1V$ and $V \leftarrow V-$ in $L_1$ is the same as that of the corresponding instructions in $L$: $V \leftarrow V+1$ and $V \leftarrow V-1$.

And obviously, the condition $V \text{ ENDS } s_1$ in $L_1$ is equivalent to the condition $V \neq 0$ in $L$.

Since $s_1$ is the only symbol in $L_1$, ending in $s_1$ is equivalent to being different from the null string.

Theorem 3.2: If a function is partially computable, then it is also partially computable in $L_n$ for each $n > 0$.

Proof:

Let the function $f$ be computed by a program $P$ in the language $L$.

We translate $P$ into a program in $L_n$ by replacing each instruction of $P$ by a macro in $L_n$.

We replace each

• instruction $V \leftarrow V+1$ by the macro $V \leftarrow V+1$,
• instruction $V \leftarrow V-1$ by the macro $V \leftarrow V-1$, and
• instruction IF $V \neq 0$ GOTO L by the macro IF $V \neq 0$ GOTO L

We are using the fact that, as proven before, $x + 1$ and $x - 1$ are both computable in base $n$, and so they can be used to define a macro in $L_n$.

Obviously, the new program computes in $L_n$ the same function $f$ that $P$ computes in $L$.

Post-Turing Programs

We will now introduce the Post-Turing language $\mathcal{T}$, which is yet another language for string manipulation. This language is named after its inventor, Emil Post, and Alan Turing, who first analyzed computation processes of this kind.

Its main difference to $L_n$ is that $\mathcal{T}$ does not use any variables.

Instead, it uses an infinite tape to store and read symbols.

This tape is divided into squares, each of which can hold a single symbol.
Post-Turing Programs

You can think of Post-Turing programs being interpreted by a machine that uses a tapehead to read and write symbols.

At any given moment during the computation, this tapehead is positioned on one definite square. Each step of a computation is sensitive to only one symbol on the tape, namely the one under the tapehead.

The $T$ programs that control the tapehead are written in a way similar to $L$ programs.

Instruction Interpretation

PRINT $\sigma$ Replace the symbol under the tapehead by the symbol $\sigma$.

IF $\sigma$ GOTO L If the symbol under the tapehead is $\sigma$, then go to the first instruction in the program labeled L; otherwise, proceed with the next instruction.

RIGHT Move the tapehead one position to the right.

LEFT Move the tapehead one position to the left.

How do we execute the computation by our programs? First of all, in addition to our alphabet $A = \{s_1, \ldots, s_n\}$, we introduce another symbol $s_0$, which serves as a punctuation mark. This symbol is called the blank and is often written as the letter $B$.

Initially, all squares of the tape contain the blank, except for a finite section that contains the input to the program.

Example 1: We want to compute a function $f(x_1, x_2, x_3)$, where $A = \{s_1, s_2\}$, $x_1 = s_1$, $x_2 = s_2s_1s_1$, and $x_3 = s_2s_2$.

Then the initial tape configuration is as follows:

```
B s_1 B s_2 s_1 B s_2 s_2
```

Example 2: We want to compute a function $f(x_1, x_2, x_3)$, where $A = \{s_1, s_2\}$, $x_1 = s_1$, $x_2 = 0$, and $x_3 = s_2s_2$.

Then the initial tape configuration is as follows:

```
B s_1 B B B B s_2 s_1 B s_2 B B B B B B ...
```

We speak of a tape configuration as consisting of the tape contents and the position of the tapehead.

Now, if we want to compute a partial function $f(x_1, \ldots, x_m)$ of $m$ variables on $A^*$, we have to place the strings $x_1, \ldots, x_m$ on the tape initially.

We do this by separating the strings by single blanks. The initial tape configuration looks like this:

```
B x_1 B x_2 B x_3 \ldots B x_m
```

So the symbol initially scanned is the blank immediately to the left of $x_1$.
Post-Turing Programs

Example 3: We want to compute a function \( f(x_1, x_2, x_3) \), where \( A = \{s_1, s_2\} \), \( x_1 = 0 \), \( x_2 = s_2s_1s_2 \), and \( x_3 = 0 \). Then the initial tape configuration is as follows:

\[
\begin{array}{c}
B & B & s_2 & s_1 & s_2 & B \\
\uparrow
\end{array}
\]

Notice that it is impossible to distinguish this initial tape configuration from that with two inputs \( x_1 = 0 \) and \( x_2 = s_2s_1s_2 \). Therefore, the number of arguments must be provided externally.