The set of primitive recursive functions is remarkably robust. But we will be able to do even more once we have developed an adequate means of handling sequences. We will identify finite sequences of natural numbers with natural numbers in the following way: the sequence \( \langle a_0, a_1, a_2, \ldots, a_k \rangle \) corresponds to the number 
\[ p_0^{a_0+1} \cdot p_1^{a_1+1} \cdot p_2^{a_2+1} \cdots p_k^{a_k+1}. \]
We add one to the exponents to guarantee that, for example, the sequences \( \langle 2, 7, 3 \rangle \) and \( \langle 2, 7, 3, 0, 0 \rangle \) have distinct numeric codes. We can take both 0 and 1 to code the empty sequence; for concreteness, let \( \Lambda \) denote 0.
Let us define the following functions:

1. \( \text{len}(s) \), which returns the length of the sequence \( s \): Let \( R(i, s) \) be the relation defined by 
   \[ R(i, s) \text{ iff } p_i \mid s \land (\forall j < s) (j > i \rightarrow p_j \not\mid s) \]
   \( R \) is primitive recursive. Now let 
   \[ \text{len}(s) = \begin{cases} 
   0 & \text{if } s = 0 \text{ or } s = 1 \\
   1 + (\min i < s) R(i, s) & \text{otherwise} 
   \end{cases} \]
   Note that we need to bound the search on \( i \); clearly \( s \) provides an acceptable bound.

2. \( \text{append}(s, a) \), which returns the result of appending \( a \) to the sequence \( s \):
   \[ \text{append}(s, a) = \begin{cases} 
   2^{a+1} & \text{if } s = 0 \text{ or } s = 1 \\
   s \cdot p_{\text{len}(s)}^{a+1} & \text{otherwise} 
   \end{cases} \]

3. \( \text{element}(s, i) \), which returns the \( i \)th element of \( s \) (where the initial element is called the 0th), or 0 if \( i \) is greater than or equal to the length of \( s \):
   \[ \text{element}(s, i) = \begin{cases} 
   0 & \text{if } i \geq \text{len}(s) \\
   \min j < s (p_j^{i+2} \not\mid s) - 1 & \text{otherwise} 
   \end{cases} \]

Instead of using the official names for the functions defined above, we introduce a more compact notation. We will use \( (s)_i \) instead of \( \text{element}(s, i) \), and \( \langle s_0, \ldots, s_k \rangle \) to abbreviate
\[ \text{append}(\text{append}(\ldots \text{append}(\Lambda, s_0) \ldots), s_k). \]
Note that if \( s \) has length \( k \), the elements of \( s \) are \( (s)_0, \ldots, (s)_{k-1} \).
It will be useful for us to be able to bound the numeric code of a sequence in terms of its length and its largest element. Suppose \( s \) is a sequence of length
$k$, each element of which is less than equal to some number $x$. Then $s$ has at most $k$ prime factors, each at most $p_{k-1}$, and each raised to at most $x + 1$ in the prime factorization of $s$. In other words, if we define

$$\text{sequenceBound}(x, k) = p_{k-1}^{k(x+1)},$$

then the numeric code of the sequence $s$ described above is at most $\text{sequenceBound}(x, k)$.

Having such a bound on sequences gives us a way of defining new functions using bounded search. For example, suppose we want to define the function $\text{concat}(s, t)$, which concatenates two sequences. One first option is to define a “helper” function $\text{hconcat}(s, t, n)$ which concatenates the first $n$ symbols of $t$ to $s$. This function can be defined by primitive recursion, as follows:

$$\text{hconcat}(s, t, 0) = s$$
$$\text{hconcat}(s, t, n + 1) = \text{append}(\text{hconcat}(s, t, n), (t)_n)$$

Then we can define $\text{concat}$ by

$$\text{concat}(s, t) = \text{hconcat}(s, t, \text{len}(t)).$$

But using bounded search, we can be lazy. All we need to do is write down a primitive recursive specification of the object (number) we are looking for, and a bound on how far to look. The following works:

$$\text{concat}(s, t) = (\min v < \text{sequenceBound}(s + t, \text{len}(s) + \text{len}(t)))$$

$$\text{len}(v) = \text{len}(s) + \text{len}(t) \land$$

$$(\forall i < \text{len}(s)) \ ((v)_i = (s)_i) \land$$

$$(\forall j < \text{len}(t)) \ ((v)_{\text{len}(s)+j} = (t)_j))$$

We will write $s \rhd t$ instead of $\text{concat}(s, t)$.

**Problem rec.1.** Show that there is a primitive recursive function $\text{sconcat}(s)$ with the property that

$$\text{sconcat}((s_0, \ldots, s_k)) = s_0 \rhd \ldots \rhd s_k.$$

**Photo Credits**

**Bibliography**