Recursive Functions

Recursive functions are built up from basic functions by some operations.
Let’s get very primitive. Suppose we have 0 defined, and want to build the nonnegative integers and our entire number system.

We define the \textit{successor} operator: the function \( S(x) \) that takes a number \( x \) to its successor \( x + 1 \).

This gives one the nonnegative integers \( \mathbb{N}_0 = \{0, 1, 2, \ldots\} \).
Defining Addition

Addition must be defined in terms of the successor function, since initially that is all we have:

\[ add(x, 0) = x \]
\[ add(x, S(y)) = S(add(x, y)) \]

For example, one can show that \( 2 + 2 = 4 \):  
\[
add(2, 2) = S(add(2, 1)) \\
= S(S(add(2, 0))) \\
= S(S(2)) \\
= S(3) \\
= 4
\]
The Three Basic Functions

We formalize the above process. Primitive recursive functions are built up from three basic functions using two operations. The basic functions are:

1. **Zero**. \( Z(x) \equiv 0 \).

2. **Successor**. \( S(x) \equiv x + 1 \).

3. **Projection**. A projection function selects out one of the arguments. Specifically

\[
P_1(x, y) \equiv x \quad \text{and} \quad P_2(x, y) \equiv y
\]
The Composition Operation

There are two operations that make new functions from old: composition and primitive recursion.

**Composition** replaces the arguments of a function by another. For example, one can define a function $f$ by

$$f(x, y) = g(h_1(x, y), h_2(x, y))$$

where one supplies the functions $g_1$, $g_2$ and $h$. 
A typical use of \textit{primitive recursion} has the following form:

\begin{align*}
    f(x, 0) &= g_1(x) \\
    f(x, S(y)) &= h(g_2(x, y), f(x, y))
\end{align*}

where one supplies the functions $g_1$, $g_2$ and $h$.

For example, in the case of addition, the $h$ is the successor function of the projection of the 2nd argument.
A special case of primitive recursion is for some constant number $k$:

\[
\begin{align*}
  f(0) &= k \\
  f(S(y)) &= h(y, f(y))
\end{align*}
\]

**Primitive recursive functions.** A function is primitive recursive if it can be built up using the base functions and the operations of composition and primitive recursion.
Composition and primitive recursion preserve the property of being computable by a TM. Thus:

**Fact.** A primitive recursive function is T-computable.
**Example: Multiplication**

\[
\text{mul}(x, 0) = 0 \\
\text{mul}(x, S(y)) = \text{add}(x, \text{mul}(x, y))
\]

(Now that we have shown addition and multiplication are primitive recursive, we will use normal arithmetical notation for them.)
Subtraction is harder, as one needs to stay within $\mathbb{N}_0$. So define “subtract as much as you can”, called \textit{monus}, written $\dot{-}$ and defined by:

$$x \dot{-} y = \begin{cases} 
  x - y & \text{if } x \geq y, \\
  0 & \text{otherwise}. 
\end{cases}$$

To formulate monus as a primitive recursive function, one needs the concept of predecessor.
Example: Predecessor

\[ \text{pred}(0) = 0 \]
\[ \text{pred}(S(y)) = y \]
Show that monus is primitive recursive.
Solution to Practice

\[
\text{monus}(x, 0) = x
\]

\[
\text{monus}(x, S(y)) = \text{pred}(\text{monus}(x, y))
\]
Example: Predicates

A function that takes on only values 0 and 1 can be thought of as a **predicate**, where 0 means false, and 1 means true.

Example: A zero-recognizer function is 1 for argument 0, and 0 otherwise:

\[
\text{sgn}(0) = 1 \\
\text{sgn}(S(y)) = 0
\]
Example: Definition by Cases

\[ f(x) = \begin{cases} 
  g(x) & \text{if } p(x), \\
  h(x) & \text{otherwise.} 
\end{cases} \]

We claim that if \( g \) and \( h \) are primitive recursive functions, then \( f \) is primitive recursive too. One way to see this is to write some algebra:

\[ f(x) \equiv g(x) \, p(x) + (1 - p(x)) \, h(x) \]
Practice

Show that if \( p(x) \) and \( q(x) \) are primitive recursive predicates, then so is \( p \land q \) (the \textit{and} of them) defined to be true exactly when both \( p(x) \) and \( q(x) \) are true.
Solution to Practice

\[ p \land q = p(x) \times q(x) \]
Functions that are not Primitive Recursive

**Theorem.** Not all T-computable functions are primitive recursive.

Yes, it’s a diagonalization argument. Each partial recursive function is given by a finite string. Therefore, one can number them $f_1, f_2, \ldots$. Define a function $g$ by

$$g(x) = f_x(x) + 1.$$ 

This $g$ is a perfectly computable function. But it cannot be primitive recursive: it is different from each primitive recursive function.
**Ackermann’s Function**

*Ackermann’s function* is a famous function that is not primitive recursive. It is defined by:

\[
A(0, y) = y + 1 \\
A(x, 0) = A(x - 1, 1) \\
A(x, y + 1) = A(x - 1, A(x, y))
\]

Here are some tiny values of the function:

\[
A(1, 0) = A(0, 1) = 2 \\
A(1, 1) = A(0, A(1, 0)) = A(0, 2) = 3 \\
A(1, 2) = A(0, A(1, 1)) = A(0, 3) = 4 \\
A(2, 0) = A(1, 1) = 3 \\
A(2, 1) = A(1, A(2, 0)) = A(1, 3) = A(0, A(1, 2)) = A(0, 4) = 5
\]
Calculate $A(2, 2)$. 
Solution to Practice

\[ A(2, 2) = A(1, A(2, 1)) = A(1, 5) = A(0, A(1, 4)). \]

Now, \[ A(1, 4) = A(0, A(1, 3)), \] and \[ A(1, 3) = A(0, A(1, 2)) = A(0, 4) = 5. \]

So \[ A(1, 4) = 6, \] and \[ A(2, 2) = 7. \]
Suppose $q(x, y)$ is some predicate. One operation is called *bounded minimization*. For some fixed $k$:

$$f(x) = \min \{ y \leq k : q(x, y) \}$$

Note that one has to deal with those $x$ where there is no $y$.

Actually, bounded minimization is just an extension of the case statement (equivalent to $k-1$ nested case statements), and so if $f$ is formed by bounded minimization from a primitive recursive predicate, then $f$ is primitive recursive.
We define

\[ f(x) = \mu q(x, y) \]

to mean that \( f(x) \) is the minimum \( y \) such that the predicate \( q(x, y) \) is true (and 0 if \( q(x, y) \) is always false).

**Definition.** A function is \( \mu \)-recursive if it can be built up using the base functions and the operations of composition, primitive recursion and unbounded minimization.
It is not hard to believe that all such functions can be computed by some TM. What is a much deeper result is that every TM function corresponds to some $\mu$-recursive function:

**Theorem.** A function is T-computable if and only if it is $\mu$-recursive.

We omit the proof.
Summary

A primitive recursive function is built up from the base functions zero, successor and projection using the two operations composition and primitive recursion. There are T-computable functions that are not primitive recursive, such as Ackermann’s function.