Chapter 3: An Introduction to Recursion Theory

**Recursive Function Theory** uses
- Simple initial functions
- Forms for building complicated functions out of simpler functions

**Primitive recursive functions:**
- 3 kinds of initial functions
- 2 forms for building functions

**Partial recursive functions:**
- All primitive recursive functions, plus
- 1 more form for building functions

Most everyday math functions are primitive recursive.

The partial recursive functions are exactly the Turing-computable functions.

(Proved at end of chapter.)
Primitive Recursive Functions

Tools for building primitive recursive functions:

- Successor function
- Constant-0 function
- Projection functions
- Function composition
- Primitive recursion
**Successor function**

The successor function, $\text{succ}(n)$ returns the successor of $n$. Defined for $n \geq 0$.

- $\text{succ}(0)$ returns 1.
- $\text{succ}(1)$ returns 2.
- $\text{succ}(57)$ returns 58.

etc.

**Constant-0 functions**

$C^k_0$ is the $k$-ary function (i.e., function with $k$ arguments) that returns 0, no matter what its arguments are. Defined for $k \geq 0$, with arguments being natural numbers.

- $C^2_0(3, 5) = 0$.
- $C^1_0(7) = 0$.
- $C^4_0(10, 9, 8, 7) = 0$.

**Projection functions**

A projection function $p^k_j$, for $k \geq 1$ and $1 \leq j \leq k$, is a $k$-ary function that returns its $j^{th}$ argument.

- $p^5_3(5, 10, 15, 20, 25) = 15$.
- $p^1_1(7) = 7$.
- $p^4_1(10, 9, 8, 7) = 10$.
**Function composition**

**Definition:**
Given functions

\[ f : N^m \rightarrow N, \text{ with } m \geq 1 \]

and \( g_1, g_2, \ldots, g_m, \text{ each } N^k \rightarrow N, \text{ with } k \geq 0, \)

\[ \text{Comp}[f, g_1, \ldots, g_m] \] means the function \( h : N^k \rightarrow N \) defined thus:

\[ h(n_1, \ldots, n_k) = f(g_1(n_1, \ldots, n_k), g_2(n_1, \ldots, n_k), \ldots, g_m(n_1, \ldots, n_k)) \]

**Idea:**
Functions \( g_1, g_2, \ldots, g_m \) are applied to the arguments; \( f \) is applied to the results returned by all the \( g \)s.

\[
\begin{align*}
n_1, n_2, \ldots, n_k & \rightarrow g_1 \rightarrow \\
n_1, n_2, \ldots, n_k & \rightarrow g_2 \rightarrow f \\

\end{align*}
\]

The \( g \)s are allowed to ignore any of their arguments, but they have all the \( n \)s available to them if they need them.

The simple function composition scheme \( f(g(n)) \) is the special case \( \text{Comp}[f, g] \), and \( k = 1 \).

Similarly, \( f(g(n_1, n_2)) \) is \( \text{Comp}[f, g] \), and \( k = 2 \).

\((m = 1 \text{ in both cases — } m \text{ is the arity of } f.)\)
Examples:

1. If $g(n) = \text{def } n^2 + 1$ and $f(n) = \text{def } 10n$, then $\text{Comp}[f, g](5)$
   
   $= f(g(5))$
   
   $= 10(5^2 + 1)$
   
   $= 260$

2. If $f(x, y) = \text{def } x + y$, then $\text{Comp}[f, p^3_1, p^3_2](20, 30, 40)$

   $= f(p^3_1(20, 30, 40), p^3_2(20, 30, 40))$

   $= p^3_1(20, 30, 40) + p^3_2(20, 30, 40)$

   $= 20 + 30 = 50$

Use: Combinations of projection functions with Comp can be used when we have "too many" arguments for the function we want to compute. If $f$ (here, $+$) has been defined, and we need to apply it to only a selection of the arguments, Comp with projection functions can apply $f$ only to the arguments we’re interested in.
**Primitive recursion**

**Definition:**

Given functions

\[ f: \mathbb{N}^k \rightarrow \mathbb{N}, \text{ with } k \geq 0 \]

and \( g^{k+2} \)

\( \text{Pr}[f, g] \) means the function \( h: \mathbb{N}^{k+1} \rightarrow \mathbb{N} \) defined thus:

\[
\begin{align*}
    h(n_1, \ldots, n_k, 0) &= f(n_1, \ldots, n_k) \\
    h(n_1, \ldots, n_k, m+1) &= g(n_1, \ldots, n_k, m, h(n_1, \ldots, n_k, m))
\end{align*}
\]

**Idea:**

\( h \) is defined recursively. The base case is \( m = 0 \), and the recursive case describes how to compute the \( m+1 \) case from the \( m \) case.

Thus, you know how to compute the 0\(^{th} \) value;
you can compute the 1\(^{st} \) value from the 0\(^{th} \) value;
you can compute the 2\(^{nd} \) value from the 1\(^{st} \) value;
and so on.

\( h(n_1, \ldots, n_k, m+1) \) is allowed to use the \( n_s \) and \( m \), if helpful,
and uses a function \( g \) to put it all together.

Sometimes only some of the arguments are useful to \( h \). \( h \) may not have to use \( m \) or all the \( n_s \) except in the recursive call. In that case, we’ll use projection functions and Comp to pick out the useful arguments.
Example: Pr[C_5^2, Comp[succ, p_3^4]] (10, 20, 30)

What are the pieces of the definition?

\[ f = C_5^2 \quad \text{Used in base case} \]
\[ g = \text{Comp}[\text{succ, } p_3^4] \quad \text{Used in recursive case} \]
\[ k = 2 \quad \text{Number of arguments to } f \]

So, what function is being defined?

\[ h(n_1, n_2, 0) = C_5^2(n_1, n_2) = 5 \]
\[ h(n_1, n_2, m+1) = \text{Comp}[\text{succ, } p_3^4](n_1, n_2, m, h(n_1, n_2, m)) = \text{succ}(p_3^4(n_1, n_2, m, h(n_1, n_2, m))) = \text{succ}(m) = m + 1 \]

Thus:

\[ h(n_1, n_2, 0) = 5 \]
\[ h(n_1, n_2, m+1) = m + 1 \]

What do we get when we evaluate \( h(10, 20, 30) \)?

\[ h(10, 20, 30) = h(10, 20, 29 + 1) = 29 + 1 = 30 \]
More examples

1. If $h = \Pr[\text{Comp}[\text{succ}, p_1]], [\text{Comp}[\text{succ}, p_2]]$:
   a) Evaluate $h(25, 36)$
   b) Evaluate $h(16, 0)$

2. If $h = \Pr[\text{Comp}[\text{succ}, p_1]], [\text{Comp}[\text{succ}, p_2]]$:
   (Note: the last projection function is different.)
   a) Evaluate $h(4, 0)$
   b) Evaluate $h(4, 3)$
**Primitive Recursive Functions: Definition**

The **primitive recursive functions** are defined as follows:

1) The unary successor function $\text{succ}$ is primitive recursive.

2) The $k$-ary constant-0 functions $C_{0}^{k}$, for each $k \geq 0$, are primitive recursive.

3) The projection functions $p_{j}^{k}$, for each $k \geq 1$ and each $1 \leq j \leq k$, are primitive recursive.

4) If $g_{1},...,g_{m}$ are $k$-ary primitive recursive functions for some $k \geq 0$, and $f$ is an $m$-ary primitive recursive function for some $m \geq 1$, then $\text{Comp}[f, g_{1},...,g_{m}]$ is a $k$-ary primitive recursive function.

5) If $f$ is a $k$-ary primitive recursive function, and $g$ is a $(k+2)$-ary primitive recursive function for some $k \geq 0$, then $\text{Pr}[f, g]$ is a $(k+1)$-ary primitive recursive function.

No other functions are primitive recursive.
Primitive Recursive Functions: Examples

(Most of these examples are taken from the book. However, they’re complex enough to merit in-class explanations.)

A strategy note: Usually, the hard part is fitting the function definition into the format imposed by Comp and/or Pr.

Pr is especially hard because it has so many arguments.
You have to specify "n"s, m, and a recursive call, even if they aren’t needed for the calculation.
So we’ll often be throwing in extra arguments, and then using projection functions to ignore some of them.

Example: f(n) = 1 is primitive recursive

\[ C_1^1(n) = \text{succ}(C_0^1(n)) \]
\[ C_1^1 = \text{Comp}[\text{succ}, C_0^1] \]

Example: f(n) = 2 is primitive recursive

\[ C_2^1(n) = \text{succ}(C_1^1(n)) \]
\[ C_2^1 = \text{Comp}[\text{succ}, C_1^1] \]

(C_1^1 was shown to be primitive recursive in the previous exercise. In general, once we show a function to be primitive recursive, we will use it freely in later constructions.)

Similar constructions show that all constant functions are primitive recursive.
(To be rigorous, use a mathematical induction proof for the general claim.)
Example: Addition is primitive recursive

Use recursive definition

\[
\begin{align*}
\text{plus}(n, 0) &= n \\
\text{plus}(n, m + 1) &= \text{succ}(\text{plus}(n, m))
\end{align*}
\]

Goal: make this fit the Pr format:

\[
\begin{align*}
h(n_1, \ldots, n_k, 0) &= f(n_1, \ldots, n_k) \\
h(n_1, \ldots, n_k, m + 1) &= g(n_1, \ldots, n_k, m, h(n_1, \ldots, n_k, m))
\end{align*}
\]

\(h\) is \(\text{plus}\).

\(f\) will be a projection function, since it returns its argument \(n\).

\[
\begin{align*}
\text{plus}(n, 0) &= p^1_1(n)
\end{align*}
\]

That is, \(f\) is \(p^1_1\).

For \(g\), rewrite \(\text{succ}(\text{plus}(n, m))\) so that it’s a function of \(n, m, \text{plus}(n, m)\):

\[
\begin{align*}
\text{succ}(\text{plus}(n, m)) \\
= \text{succ}(p^3_3(n, m, \text{plus}(n, m))) \\
= \text{Comp}[\text{succ}, p^3_3](n, m, \text{plus}(n, m))
\end{align*}
\]

That is, \(g = \text{Comp}[\text{succ}, p^3_3]\)

Putting this together, we have

\[
\begin{align*}
\text{plus}(n, 0) &= p^1_1(n) \\
\text{plus}(n, m + 1) &= \text{Comp}[\text{succ}, p^3_3](n, m, \text{plus}(n, m))
\end{align*}
\]

That is, \(\text{plus} = \text{Pr}[p^1_1, \text{Comp}[\text{succ}, p^3_3]]\)
Example: \( \text{mult}(n, m) \) is primitive recursive

Use recursive definition

\[
\begin{align*}
\text{mult}(n, 0) &= 0 \\
\text{mult}(n, m + 1) &= n + \text{mult}(n, m) \\
&= \text{plus}(n, \text{mult}(n, m))
\end{align*}
\]

Goal: make this fit the \textbf{Pr} format:

\[
\begin{align*}
h(n_1, \ldots, n_k, 0) &= f(n_1, \ldots, n_k) \\
h(n_1, \ldots, n_k, m + 1) &= g(n_1, \ldots, n_k, m, h(n_1, \ldots, n_k, m))
\end{align*}
\]

\( h \) is \( \text{mult} \).

\( f \) will be a constant-0 function:

\[
\begin{align*}
\text{mult}(n, 0) &= C^1_0(n) \\
\end{align*}
\]

That is, \( f = C^1_0 \).

For \( g \), rewrite \( \text{plus}(n, \text{mult}(n, m)) \) to be a function of \( n, m, \text{plus}(n, m) \):

\[
\begin{align*}
\text{plus}(n, \text{mult}(n, m)) &= \text{plus}(p^3_1(n, m, \text{mult}(n, m)), p^3_3(n, m, \text{mult}(n, m))) \\
&= \text{Comp}[\text{plus}, p^3_1, p^3_3](n, m, \text{mult}(n, m))
\end{align*}
\]

That is, \( g = \text{Comp}[\text{plus}, p^3_1, p^3_3] \)

Putting this together, we have

\[
\begin{align*}
\text{mult}(n, 0) &= C^1_0(n) \\
\text{mult}(n, m + 1) &= \text{Comp}[\text{plus}, p^3_1, p^3_3](n, m, \text{mult}(n, m))
\end{align*}
\]

That is, \( \text{mult} = \text{Pr}[C^1_0, \text{Comp}[\text{plus}, p^3_1, p^3_3]] \)
Example: \( \text{pred}(n) \) is primitive recursive

Use a definition by cases:

\[
\text{pred}(n) = 0 \quad \text{if } n = 0 \\
\text{pred}(n) = n - 1 \quad \text{otherwise}
\]

or, alternatively,

\[
\text{pred}(0) = 0 \\
\text{pred}(m + 1) = m
\]

Goal: make this fit the \( \text{Pr} \) format:

\[
\begin{align*}
\text{h}(n_1, \ldots, n_k, 0) &= \text{f}(n_1, \ldots, n_k) \\
\text{h}(n_1, \ldots, n_k, m + 1) &= \text{g}(n_1, \ldots, n_k, m, \text{h}(n_1, \ldots, n_k, m))
\end{align*}
\]

Why \( \text{Pr} \), when the definition of \( \text{pred} \) isn’t recursive?

Because \( \text{Pr} \) is a way to get a choice between cases.

\( h \) is \( \text{pred} \).

\( f \) will be a constant-0 function:

\[
\text{pred}(0) = C_0^0( )
\]

\( g \) uses a projection function to get \( m \) — we ignore the actual recursive call.

\[
\text{pred}(m + 1) = p_1^2(m, \text{pred}(m))
\]

Putting this together, we have

\[
\begin{align*}
\text{pred}(0) &= C_0^0( ) \\
\text{pred}(m + 1) &= p_1^2(m, \text{pred}(m))
\end{align*}
\]

That is, \( \text{pred} = \text{Pr}[C_0^0, p_1^2] \)
Example: *monus*(n, m) is primitive recursive

Definition of monus:

\[
\begin{align*}
\text{monus}(n, m) &= 0 & \text{if } n < m \\
\text{monus}(n, m) &= n - m & \text{if } n \geq m
\end{align*}
\]

Use this recursive definition:

\[
\begin{align*}
\text{monus}(n, 0) &= n \\
\text{monus}(n, m + 1) &= \text{pred}(\text{monus}(n, m))
\end{align*}
\]

Goal: make this fit the Pr format:

\[
\begin{align*}
h(n_1, \ldots, n_k, 0) &= f(n_1, \ldots, n_k) \\
h(n_1, \ldots, n_k, m + 1) &= g(n_1, \ldots, n_k, m, h(n_1, \ldots, n_k, m))
\end{align*}
\]

h is monus.

f is a projection function which returns n:

\[
\text{monus}(n, 0) = p_{1}^{1}(n)
\]

g composes pred with a projection function.

\[
\begin{align*}
\text{monus}(n, m + 1) &= \text{pred}(\text{monus}(n, m)) \\
&= \text{pred}(p_{3}^{3}(n, m, \text{monus}(n, m))) \\
&= \text{Comp}[\text{pred}, p_{3}^{3}] (n, m, \text{monus}(n, m)))
\end{align*}
\]

Putting this together, we have

\[
\begin{align*}
\text{monus}(n, 0) &= p_{1}^{1}(n) \\
\text{monus}(n, m + 1) &= \text{Comp}[\text{pred}, p_{3}^{3}] (n, m, \text{monus}(n, m))
\end{align*}
\]

That is, \( \text{monus} = \text{Pr}[p_{1}^{1}, \text{Comp}[\text{pred}, p_{3}^{3}]] \)
A polynomial example: \( 2n + 3 \) is primitive recursive

\[
2n + 3 = \text{plus}(\text{mult}(2, n), 3) \\
= \text{plus}(\text{mult}(C_2^1(n), p_1^1(n)), C_3^1(n)) \\
= \text{plus}(\text{Comp}[\text{mult}, C_2^1, p_1^1](n), C_3^1(n)) \\
= \text{Comp}[\text{plus, Comp}[\text{mult}, C_2^1, p_1^1], C_3^1](n)
\]

So this polynomial function is \( \text{Comp}[\text{plus, Comp}[\text{mult}, C_2^1, p_1^1], C_3^1] \).
\( sg(n) \) is primitive recursive

Definition:

\[
sg(n) = \begin{cases} 
0 & \text{if } n = 0 \\
1 & \text{otherwise}
\end{cases}
\]

(This will be a useful function in later discussions.)

\[
sg(0) = 0 = C_0^0( ) \\
sg(m + 1) = C_1^2(m, sg(m))
\]

Thus \( sg = \Pr[C_0^0, C_1^2] \)

\( \overline{sg}(n) \) is primitive recursive

Definition:

\[
\overline{sg}(n) = \begin{cases} 
1 & \text{if } n = 0 \\
0 & \text{otherwise}
\end{cases}
\]

(This will also be a useful function in later discussions.)

\[
\overline{sg}(0) = 1 = C_1^0( ) \\
\overline{sg}(m + 1) = C_0^2(m, \overline{sg}(m))
\]

Thus \( \overline{sg} = \Pr[C_1^0, C_0^2] \)
**Primitive Recursive Predicates**

A predicate is an expression that is either true or false.

**Examples:**

- `prime(n)`
- `greater_than(x, y)`

Predicates are

- Usually written in prefix **notation**: `prime(n)`
- Sometimes written in infix **notation**: `x > y`

In this text, predicates will always be defined for all natural-number arguments; there are no "partial predicates" analogous to "partial functions".

The **characteristic function of a predicate** returns 1 if the predicate is true and 0 if the predicate is false.

More formally, if \( C(n_1, n_2, \ldots, n_k) \) is a predicate, then its characteristic function \( \chi_C \) is defined as:

\[
\chi_C(n_1, n_2, \ldots, n_k) = \begin{cases} 
1 & \text{if } C(n_1, n_2, \ldots, n_k) \\
0 & \text{otherwise}
\end{cases}
\]

Since all predicates are "total", \( \chi_C \) is a total function.
**Definition:** A **primitive recursive predicate** is a predicate with a primitive recursive characteristic function.

(We use characteristic functions to apply the idea of "primitive recursive" to predicates.)

**Example:** greater(n, m) is a primitive recursive predicate.

Characteristic function:

\[
\chi_{\text{greater}}(n, m) = \begin{cases} 
1 & \text{if } n > m \\
0 & \text{otherwise}
\end{cases}
\]

\[
\chi_{\text{greater}}(n, m) = \text{sg}(\text{monus}(n, m)) = \text{Comp}[\text{sg, monus}](n, m)
\]

Similarly, gr_eq(n, m) and eq(n, m) are primitive recursive:

\[
\chi_{\text{gr_eq}}(n, m) = \overline{\text{sg}}(\text{monus}(m, n))
\]

\[
\chi_{\text{eq}}(n, m) = \overline{\text{sg}}(\text{plus}(\text{monus}(n, m), \text{monus}(m, n)))
\]

The other relational operators (<, ≤, ≠) can be constructed from these using the logical operators discussed below.
**New convention:** In most examples from here on, we won’t take the time to reduce all function definitions to canonical notation.

We’ll use standard math notation.

We assume that we could translate if we needed to.

**Example:** The logical connectives are primitive recursive

\[
\& \quad \chi_{C_1 \& C_2}(\vec{n}) = \chi_{C_1}(\vec{n}) \ast \chi_{C_2}(\vec{n})
\]

\[
\neg \quad \chi_{\neg C}(\vec{n}) = \overline{sg}(\chi_{C}(\vec{n}))
\]

\[
\lor \quad \chi_{C_1 \lor C_2}(\vec{n}) = sg(\chi_{C_1}(\vec{n}) + \chi_{C_2}(\vec{n}))
\]

(Alternatively, \( \chi_{C_1 \lor C_2}(\vec{n}) = \chi_{\neg(\neg C_1 \& \neg C_2)}(\vec{n}) \))

\[
\rightarrow, \leftrightarrow \quad \text{These can be constructed from } \&, \lor, \text{ and } \neg.
\]

**Notation:** \( \vec{n} = n_1, n_2, \ldots, n_k \)
Example: A function defined by cases using primitive recursive predicates is primitive recursive.

If a function $f$ is defined this way:

$$f(n_1, n_2, \ldots, n_k) = \begin{cases} 
  g_1(n_1, \ldots, n_k) & \text{if } C_1(n_1, n_2, \ldots, n_k) \\
  g_2(n_1, \ldots, n_k) & \text{if } C_2(n_1, n_2, \ldots, n_k) \\
  \vdots \\
  g_m(n_1, \ldots, n_k) & \text{if } C_m(n_1, n_2, \ldots, n_k)
\end{cases}$$

where

- all the $g$s are primitive recursive functions,
- all the $C$s are primitive recursive predicates, and
- exactly one $C$ holds for any $n_1, n_2, \ldots, n_k$ (the $C$s are mutually exclusive and collectively exhaustive),

then $f$ is primitive recursive.

Why?

Using notation $\vec{n} = n_1, n_2, \ldots, n_k$,

$$f(n) = g_1(\vec{n}) \chi_{C_1}(\vec{n}) + g_2(\vec{n}) \chi_{C_2}(\vec{n}) + \ldots + g_m(\vec{n}) \chi_{C_m}(\vec{n})$$
Specific example of definition by cases:

\[
f(x, y) = \begin{cases} 
0 & \text{if } x < y \\
1 & \text{if } x = y \\
2 & \text{if } x > y 
\end{cases}
\]

or, more formally,

\[
f(x, y) = \begin{cases} 
C_1^2(x, y) & \text{if } x < y \\
C_1^2(x, y) & \text{if } x = y \\
C_2^2(x, y) & \text{if } x > y 
\end{cases}
\]

\[
f(x, y) = 0 \ast \chi_<(x, y) + 1 \ast \chi_= (x, y) + 2 \ast \chi_>(x, y)
\]

Exactly one of the \( \chi \)s will be 1, with all the others 0.
A computable function that is not primitive recursive: Ackermann’s function

\[ H(0, m) = m + 1 \]
\[ H(n + 1, 0) = H(n, 1) \]
\[ H(n + 1, m + 1) = H(n, H(n + 1, m)) \]

Example calculation: \( H(2, 1) \)

\[
H(2,1) = H(1,H(2,0)) \\
= H(1,1) \\
= H(1,H(0,H(1,0))) \\
= H(1,H(0,1) + 1) \\
= H(1,2+1) \\
= H(1,3) \quad \text{Note: } H(1,1) = 3 \\
= H(0,H(1,2)) \\
= H(1,2) + 1 \\
= H(0,H(1,1)) + 1 \\
= H(0,3) + 1 \\
= 3 + 1 + 1 \\
= 5
\]
Partial Recursive Functions

New function-forming tool: **Minimization**

**Definition:** $\mu m[C(n_1, n_2, \ldots, n_k, m)]$ is the least natural number $m$ such that $C(n_1, n_2, \ldots, n_k, m)$ holds, where $C$ is any $(k+1)$-ary predicate.

**Examples:**

$$\mu m[m + 3 > 10] = 8$$
$$\mu m[m - 1 \text{ is prime}] = 3$$

**Exercise:**

$$\mu m[n_1 \text{ mod } n_2 = m](5, 2) = \text{ what?}$$
$$\mu m[m \text{ mod } n_1 = n_2](5, 2) = \text{ what?}$$
$$\mu m[n_1 \text{ mod } m = n_2](5, 2) = \text{ what?}$$
Answers to exercise:

\[ \mu m[n_1 \mod n_2 = m](5, 2) = 1 \]

5 mod 2 = 1

\[ \mu m[m \mod n_1 = n_2](5, 2) = 2 \]

2 mod 5 = 2

\[ \mu m[n_1 \mod m = n_2](5, 2) = 3 \]

5 mod 3 = 2

Copyright © 2010 Nancy Lynn Tinkham
**Definition:** Suppose we are given function $f:N^{k+1} \rightarrow N$, with $k \geq 0$. Then $\text{Mn}[f]$ means the function $g$ defined as:

$$g(n_1, n_2, \ldots, n_k) = \mu m[f(n_1, n_2, \ldots, n_k, m) = 0 \text{ and such that, for all } j < m, f(n_1, n_2, \ldots, n_k, j) \text{ is defined and } \neq 0]$$

**Example:**

Suppose $f$ has the following values:

- $f(0) = 3$
- $f(1) = 1$
- $f(2) = 4$
- $f(3) = 2$
- $f(4) = 0$
- $f(5) = 3$
- $f(6) = 1$
...

$\text{Mn}[f](\ ) = 4$ (because $f(4) = 0$)

**Example:**

Suppose $f$ has the following values:

- $f(0,0) = 5$
- $f(1,0) = 1$
- $f(2,x) = 2$ for all $x$
- $f(0,1) = 2$
- $f(1,1)$ undefined
- $f(0,2) = 6$
- $f(1,2) = 4$
- $f(0,3) = 0$
- $f(1,3) = 0$
- $f(0,4)$ undefined
- $f(1,4) = 3$
- $f(0,5) = 0$
- $f(1,5) = 10$
...

$\text{Mn}[f](0) = 3$ (because $f(0,3) = 0$)

$\text{Mn}[f](1)$ is undefined (because undefined at a value smaller than the first one producing 0)

$\text{Mn}[f](2)$ is undefined (because never 0)
Definition: Partial Recursive Functions

Definition: The class of partial recursive functions is the smallest class containing all initial functions (succ, constant-0, and projection fn) and closed under operations Comp, Pr, and Mn.

Definition: A partial recursive function that is total is called a total recursive function, or simply a recursive function.

Definition: A predicate with a partial recursive characteristic function is called a recursive predicate.

Recall:

All predicates are total.
All characteristic functions of predicates are total.

Example: div(x, y) is partial recursive

\[
\text{div}(x, y) = \mu m[\text{succ}(x) \div (m \cdot y + y) = 0] \quad \text{(Note: \(\div\) is monus)}
\]

I.e.: \(\text{div}(x, y) = \text{Mn}[f](x, y, m)\), where \(f(x, y, m) = \text{succ}(x) \div (m \cdot y + y)\)

(Note: div is not primitive recursive because it is not total.)

Specific example:

\[
\text{div}(6, 2) = \mu m[\text{succ}(6) \div (m \cdot 2 + 2) = 0] = \mu m[7 \div (m \cdot 2 + 2) = 0]
\]

\[
\begin{align*}
7 \div (0 \cdot 2 + 2) &= 7 \div 2 = 5 \\
7 \div (1 \cdot 2 + 2) &= 7 \div 4 = 3 \\
7 \div (2 \cdot 2 + 2) &= 7 \div 6 = 1 \\
7 \div (3 \cdot 2 + 2) &= 7 \div 8 = 0
\end{align*}
\]

Thus, \(\text{div}(6, 2) = 3\)

\[
\text{div}(5, 2) = \mu m[\text{succ}(5) \div (m \cdot 2 + 2) = 0] = \mu m[6 \div (m \cdot 2 + 2) = 0]
\]

\[
\begin{align*}
6 \div (0 \cdot 2 + 2) &= 6 \div 2 = 4 \\
6 \div (1 \cdot 2 + 2) &= 6 \div 4 = 2 \\
6 \div (2 \cdot 2 + 2) &= 6 \div 6 = 0
\end{align*}
\]

Thus, \(\text{div}(5, 2) = 2\)
Equivalence of Turing Machines and Partial Recursive Functions

Theorem: A number-theoretic function is partial recursive iff it is Turing-computable.

I’ll omit the proof in class, as it’s a long and detailed one; the proof is in the book, if you want to see it.

Idea of proof: Simulate a TM with a partial recursive function computation; and simulate each of the partial recursive function "building blocks" with a TM operation.

To simulate prfs with a TM, show that succ, the constant-0 functions, the projection functions, Comp, Pr, and Mn are computable on a TM.

To simulate a TM with a prf:

- Encode the tape configuration as a number:
  + Use a 1-way-infinite tape
  + Represent each tape symbol with a base-$k$ number, where $k$ is the size of $\Gamma \cup \{\text{B}\}$ (use 0 for B); represent the entire tape as a base-$k$ number by reading the tape-symbol numbers right-to-left and omitting leading 0s.

- Represent each state and each tape position with a number.

- Represent each instantaneous description of the TM with a triple of these numbers.

- Define mathematical functions that compute the next triple (literally, $2^{\text{tape contents}} \times 3^{\text{state}} \times 5^{\text{position}}$) from the current triple, for each possible TM operation (move, write, change state).

- Show that these functions are partial recursive.

Implications of this theorem:

When showing something is computable, we can use either formulation, as convenient.

Unless we’re talking about one of the computing models in particular, we’ll use the terms Turing-computable and partial recursive interchangeably.