Chapter 2

Primitive Recursion

Recall the skeptical attitude regarding foundations that followed the paradoxes in analysis and set theory. "Finitists" urged a "back-to-basics" approach based on such elementary constructions that one could be confident nothing would go wrong. The primitive recursive functions are a simple collection of intuitively computable functions that many finitists could be comfortable with. They interest us because they will serve as our "springboard" into the general theory of computable functions.

2.1 Basic functions

2.1.1 Zero function

$o(x) = 0$.

2.1.2 Successor

$s(x) = x + 1$.

2.1.3 Projection

(picking out an argument): $p^i_k(x_1, \ldots, x_k) = x_i$.

Next we consider effective ways of building new functions out of old ones. Note that these are operations on *functions*, not numbers.

2.2 Basic operations on functions

2.2.1 Composition

If f has arity m and each g_i has arity k then $C(f, g_1, \ldots, g_m)$ denotes the unique k-ary function h such that for each k-ary $\vec{x}$:

$$h(\vec{x}) = f(g_1(\vec{x}), ..., g_m(\vec{x})).$$

2.2.2 Primitive Recursion

If f has arity $k+2$ and g has arity k, then $R(g, f)$ denotes the unique $(k+1)$-ary function h such that for each k-ary $\vec{y}$:

$$\begin{aligned} h(0, \vec{y}) &= g(\vec{y}); \\ h(x+1, \vec{y}) &= f(h(x, \vec{y}), x, \vec{y}). \end{aligned}$$

In order to cover the (awkward) case in which y is empty (so g is 0-ary), we adopt the convention that the constant n is a 0-ary constant function:

$$n() = n.$$

Then in the case of unary primitive recursion:

$$\begin{aligned} h(0) &= n(); \\ h(x+1) &= f(h(x), x); \end{aligned}$$

we may write $h = R(n, f)$, where $n = n()$ is understood to be a 0-ary function and hence R is the same operation on functions as in the case of arities greater than 1. Note that this convention extends to composition. If the component functions are 0-ary, $C(f, n_1, \ldots, n_k) = f(n_1, \ldots, n_k)$. If the outer function is 0-ary n, then $C(n) = n$.

2.3 The set of primitive recursive functions

$Prim$ = the least set X such that:

1. The 0-ary functions are all in X,
2. The basic functions are in X, and
3. X is closed under C and R.

Notice that we are already looking at closing a collection under some operations. Obviously the basic functions are not closed under composition or primitive recursion. So we are looking at a hierarchy of some sort.

2.4 Primitive Recursive Derivations

Each function f in $Prim$ can be defined from the basic functions using operators C and R. We may think of the basic functions invoked as leaves in a tree whose non-terminal nodes are labelled with C and R. Nodes labelled by C may have any number of daughters and nodes labelled by R always have two daughters. We may think of this tree as a program for computing the function so defined. We will do several proofs by induction on the depth of this tree. This is similar to doing inductive proofs in logic on the depth of embedding of the logical connectives in a formula.

2.5 Primitive Recursive Relations

A relation $R(x)$ is primitive recursive just in case its characteristic function χ_R is primitive recursive:

$$\begin{aligned} \chi_R(x) &= 1 \text{ if } R(x), \\ \chi_R(x) &= 0 \text{ if } \neg R(x). \end{aligned}$$

We will simplify notation by letting the relation stand for its own characteristic function when no confusion results.

$$\chi_R(x) = R(x).$$

2.6 A Stockpile of Primitive Recursive Functions

This looks like a pretty simple programming language. But only a few primitive recursions are required to produce most functions you would ever care about. It is a beautiful thing to see how explosively it happens (cf. Cutland, p. 36).

2.6.1 Constant functions

$c_i(x) = i$.

Observe that

$$\begin{aligned} c_0(x) &= 0; \\ c_{n+1}(x) &= s(c_n(x)) \\ &= C(s, c_n)(x). \end{aligned}$$

Hence:

$$\begin{aligned} c_0(x) &= 0; \\ c_{n+1}(x) &= C(s, c_n)(x). \end{aligned}$$

Notice, this is a recursive definition of a family of primitive recursive functions $c_i(x)$, not a primitive recursive definition of a single primitive recursive function $c(i, x)$. In the lingo, a definition that puts a parameter into the argument list of a function is said to be uniform in that parameter. Thus, $c(i, x)$ is uniform in i, whereas $c_i(x)$ is not.

2.6.2 Addition

I am going to go through this example in complete detail, showing the heuristic procedure for arriving at a primitive recursive derivation of a function.

Sensible way to define it:

$$\begin{aligned} y + 0 &= y; \\ y + (x + 1) &= (y + x) + 1. \end{aligned}$$

Rewrite + and successor in prefix notation:

$$\begin{aligned} +(0, y) &= y; \\ +(x + 1, y) &= s(+(x, y)). \end{aligned}$$

This is not the required form for primitive recursion. We need a unary function $g(y)$ on the right hand side of the base case and a ternary function $f(+(x, y), x, y)$ on the right hand side in the inductive case. Compose in projections to get the argument lists in the required form:

$$\begin{aligned} +(0, y) &= p_1^1(y); \\ +(x + 1, y) &= s(p_3^1(+(x, y), x, y)). \end{aligned}$$

Now be painfully literal about composition being an operator on a fixed list of arguments to arrive at the required form for primitive recursion:

$$\begin{aligned} +(0, y) &= p_1^1(y); \\ +(x + 1, y) &= C(s, p_3^1)(+(x, y), x, y)). \end{aligned}$$

Now apply the primitive recursion operator:

$$+ = R(p_1^1, C(s, p_3^1)).$$

I'll let you verify that the following derivations work:

2.6.3 Multiplication

$$\begin{aligned} \cdot &= R(o, C(+, p_3^1)) \\ &= R(o, C(R(p_1^1, C(s, p_3^1)), p_3^1, p_3^3)). \end{aligned}$$

2.6.4 Exponentiation

How many nested primitive recursions occur in the following derivation?

$$\begin{aligned} Exp &= R(c_1, C(\cdot, p_3^1)) \\ &= R(C(s, o), C(R(0, C(R(p_1^1, C(s, p_3^1)), p_3^1)), p_3^1, p_3^3)). \end{aligned}$$

2.6.5 Decrement

$Dec(x) = 0$ if $x = 0$ and $Dec(x) = x - 1$ otherwise. We get this by a sneaky application of R.

$$\begin{aligned} Dec(0) &= 0 \\ Dec(x+1) &= x; \\ &= p_2^2(Dec(x), x). \end{aligned}$$

Thus, in light of the convention that constants are 0-ary primitive recursive functions:

$$Dec = R(0, p_2^2).$$

2.6.6 Cutoff Subtraction

This is just like addition, except that successor is replaced with decrement.

$$\begin{aligned} y \dot{-} 0 &= y; \\ y \dot{-} (x+1) &= Dec(y \dot{-} x). \end{aligned}$$

Hence:

$$\begin{aligned} \dot{-} &= R(p_1^1, C(Dec, p_3^1)) \\ &= R(p_1^1, C(R(0, p_2^2), p_3^1)). \end{aligned}$$

2.6.7 Factorial

$$\begin{aligned} 0! &= 1; \\ (x+1)! &= x! \cdot (x+1), \end{aligned}$$

so

$$! = R(1, C(\cdot, p_2^1, C(s, p_2^2)).$$

I think we have all seen enough of official derivations. From now on I will just write the obvious recurrence equations.

2.6.8 Signature

$$\begin{aligned} sg(0) &= 0; \\ sg(x+1) &= 1 \end{aligned}$$

2.6.9 Reverse signature

$$\begin{aligned}\overline{sg}(0) &= 1;\\ \overline{sg}(x+1) &= 0.\end{aligned}$$

2.6.10 Identity

$$\chi_{=}(x,y) = \overline{sg}((x \dot{-} y) + (y \dot{-} x)).$$

2.6.11 Ordering

$$\chi_{>}(x,y) = sg(x \dot{-} y).$$

2.6.12 Min

$$\min(x,y) = x \dot{-} (x \dot{-} y).$$

2.6.13 Max

$$\max(x,y) = x + (y \dot{-} x).$$

2.6.14 Absolute difference

$$|x - y| = (x \dot{-} y) + (y \dot{-} x).$$

2.6.15 Remainder when x is divided by y

Incrementing the numerator x increments the remainder until the remainder is incremented up to y, when the remainder drops to 0 because another factor of y can be taken out of x. The following program implements this idea by using sg to add 1 if incrementing the previous remainder doesn't climb all the way up to y (i.e., if the difference $|x-(rm(x,y)+1)|$ is nonzero) and 0 otherwise. Observe that in the case of division by 0, this condition is always met for nonzero y, so $x/0 = x$. This causes no harm since we never actually use the function in this case, so we can conventionally define division by 0 to be whatever we want.

$$\begin{aligned}rm(0,y) &= 0;\\ rm(x+1,y) &= (rm(x,y)+1)\cdot sg(|y-(rm(x,y)+1)|).\end{aligned}$$

Do you see how to get the messy composition into the required form $f(rm(x,y),x,y)$?

2.6.16 Quotient

$qt(x, y) =$ the greatest lower bound of y/x in the natural numbers, which is often denoted $\lceil y/x \rceil$. The idea here is that we get a bigger quotient each time the numerator is incremented to include another factor of y. This can be checked by incrementing the previous remainder and checking whether the result is y.

$$\begin{array}{rcl} qt(0, y) & = & 0; \\ qt(x + 1, y) & = & qt(x, y) + \overline{sg}(|y - (rm(x, y) + 1)|). \end{array}$$

2.6.17 Divisibility

$x|y$ is the relation "x divides y evenly".

$$x|y = \overline{sg}(rm(x, y)).$$

Exercise 2.1 *Provide full formal definitions (i.e., in terms of C, R, and the basic functions only) for three of the functions listed from signature on.*

2.7 Derived Primitive Recursive Operators

Wait a minute! The construction rules are tedious (think about using sg as a conditional branch). Wouldn't it be better to prove that $Prim$ is closed under more operators? Then we could use these as operators to construct more primitive recursive functions in a more natural way instead of always putting them into this clunky form. (That's why closure laws are so desirable— if we find more operators the collection is closed under, we have more ways of "reaching" each element of the class from basic objects.). The collection $Prim$ is closed under each of the following operators:

2.7.1 Substitution of a constant

If $f(x_1, \ldots, x_i, \ldots, x_n)$ is primitive recursive,
then so is $h(x_1, \ldots, x_n) = g(x_1, \ldots, k, \ldots, x_n)$.
The idea is that we can compose in appropriate projections and a constant function:

$$\begin{array}{rcl} h(x_1, \ldots, x_n) & = & g(p_n^1(x_1, \ldots, x_i, \ldots, x_n), \\ & & \ldots, c_k(p_n^i(x_1, \ldots, x_i, \ldots, x_n), \\ & & \ldots, p_n^n(x_1, \ldots, x_i, \ldots, x_n)). \end{array}$$

2.7.2 Sum

Let any indexed collection $\{f_z | z \in \mathbf{N}\}$ be given. Think of

$$\sum_{z \leq x} f_z(y) = h(x, y)$$

as a function of x and y. Then we may think of it also as an x-ary operation on the first x functions in the indexed set:

$$\sum_{z \leq x} f_z(y) = \sum\nolimits_x (f_0, \ldots, f_x)(y).$$

We can establish that operation $\sum_{z \leq x}$ preserves primitive recursiveness as follows:

$$\begin{aligned} \sum_{z<0} f_z(y) &= 0; \\ \sum_{z<x+1} f_z(y) &= \sum_{z<x} f_z(y) + f_x(y). \end{aligned}$$

2.7.3 Bounded Sum

The preceding construction is non-uniform in x. There is also a uniform version of bounded summation on a single function. Think of

$$g(x, y) = \sum_{z<x} f(z, y)$$

as a function of x and y.

$$\begin{aligned} \sum_{z<0} f(z, y) &= 0; \\ \sum_{z<x+1} f(z, y) &= \left[\sum_{z<x} f(z, y)\right] + f(x, y). \end{aligned}$$

2.7.4 Product

Similar to the sum case.

$$\begin{aligned} \prod_{z<0} f_z(y) &= 1; \\ \prod_{z<x+1} f_z(y) &= \prod_{z<x} f_z(y) \cdot f_x(y). \end{aligned}$$

2.7.5 Bounded Product

Similar to bounded sum.

$$\begin{aligned} \prod_{z<0} f(z, y) &= 1; \\ \prod_{z<x+1} f(z, y) &= \left[\prod_{z<x} f(z, y)\right] \cdot f(x, y). \end{aligned}$$

2.7.6 Definition by Cases

Let the P_i be mutually exclusive and exhaustive primitive recursive relations.

$$\sum_{z<k+1} g_z(x) \cdot P_z(x) = \begin{cases} g_1(x) & \text{if } P_1(x); \\ g_2(x) & \text{if } P_2(x); \\ \vdots & \\ g_k(x) & \text{if } P_k(x). \end{cases}$$

2.8 Logical Operators

2.8.1 Conjunction

$$P(x) \wedge Q(x) = P(x) \cdot Q(x).$$

2.8.2 Negation

$$\neg P(x) = \overline{sg}(P(x)).$$

2.8.3 Disjunction

$$\begin{aligned} P(x) \vee Q(x) &= \neg(\neg P(x) \wedge \neg Q(x)); \\ &= \max(P(x), Q(x)). \end{aligned}$$

2.8.4 Conditional

$$P(x) \to Q(x) = \neg P(x) \vee Q(x).$$

2.8.5 Biconditional

$$P(x) \leftrightarrow Q(x) = P(x) \to Q(x) \wedge Q(x) \to P(x).$$

2.8.6 Bounded Universal quantifier

$$\forall_{z<x} P(x, \vec{y}) = \prod_{z<x} P(z, \vec{y}).$$

2.8.7 Bounded Existential quantifier

$$\exists_{z<x} P(x, \vec{y}) = \sum_{z<x} P(z, \vec{y}).$$

2.8.8 Bounded Minimization

$$\begin{aligned} g(x, y) &= \min_{z \le x}[f(z, y) = 0]; \\ &= \text{"the least } z \le x \text{ such that } f(x, y) = 0\text{"}. \end{aligned}$$

By this we mean that if no root of f is found up to the bound, we return the bound to keep the function total. If the value returned under search bound n is strictly less than n, then a root of f was found, so we don't increment when the search bound is raised to $n + 1$. If the value returned under search bound n is identical to n, then either we (coincidentally) found a root at the last minute, or we stopped because we hit the bound. So we have to check, further, if the bound is a root in that case. Thus, we increment just in case the previous search hit the bound and the bound is not a root.

$$\begin{aligned} \min_{z \le 0}(f(z, \vec{y}) = 0 &= 0; \\ \min_{z \le x+1}(f(z, \vec{y}) = 0) &= \min_{z \le x}(f(z, \vec{y}) = 0) + \\ &\quad +[\min_{z \le x}(f(z, \vec{y}) = 0) = x \wedge f(x, \vec{y}) > 0]; \\ \min_{z \le x}(P(z, \vec{y})) &= \min_{z \le x}\overline{sg}(\chi_P(z, \vec{y})). \end{aligned}$$

2.8.9 Bounded Maximization

$$\max_{z \le x}\left(P(x, \vec{y})\right) = x \dot{-} \min_{z \le x}\left(P(x \dot{-} z, \vec{y})\right).$$

2.8.10 Iteration

$$\begin{aligned} f^0(y) &= y; \\ f^{x+1}(y) &= f(f^x(y)). \end{aligned}$$

Exercise 2.2 *Show that for each primitive recursive function there is a monotone primitive recursive function that is everywhere greater.*

Exercise 2.3

1. Prove closure of *Prim* under any one of the operators.
2. Provide full formal definitions for any three of the logical operators.

2.9 Some Number Theoretical Constructions

With our new operations we are on Easy Street. The Prime number predicate is programmed just by writing its logical definition! How's that for a handy computer language?

2.9.1 Primality

$$Prime(x) \iff 1 < x \wedge \forall_{z<x}(z = 1 \vee \neg(z|x)).$$

2.9.2 The first prime after x

This one uses Euclid's theorem that there exists at least one prime p such that for any x, $x < p \leq x! + 1$. Why is that? Well, $x! + 1$ has some prime factor $\leq x!+1$ by the uniqueness and existence of prime factorizations (the fundamental theorem of arithmetic). But $x! + 1$ has no factor $\leq x$, since each such divisor leaves remainder 1. Hence, $x! + 1$ has a prime factor $x < p \leq x! + 1$.

$$h(x) = \min_{z \leq x!+1}(Prime(z) \wedge x < z).$$

2.9.3 The x^{th} prime

$$\begin{aligned} p(0) &= 2; \\ p(x+1) &= h(p(x)). \end{aligned}$$

2.9.4 Exponent of the x^{th} prime in the prime factorization of y

How do we know it's unique? How do we know the search bound is high enough? (Show by induction that $2^y > y$. 2 is the lowest possible prime factor of y, so we're safe.)

$$[y]_x = \max_{z<y}(p(x)^z | y).$$

2.10 Gödel Numbering Finite Sequences

Let $\vec{x}$ be an n-ary sequence. Define

2.10.1 The Gödel coding of $\vec{x}$

For each number $n \geq 1$, the prime factorization of n is a finite product of form

$$\Pi_{i \leq k} p(i)^{k_i}.$$

We know from the fundamental theorem of arithmetic that the prime factorization exists and is unique, for each number $n \geq 1$. Think of the prime factorization of n as encoding a finite sequence as follows:

1. The first prime whose exponent is zero in the prime factorization determines the length of the coded sequence.

2. The exponents of earlier prime factors are one greater than the item in the corresponding position in the coded sequence.

There is just one flaw: zero encodes no sequence since each prime factorization is a product and the empty product is 1. So we will take n to encode a sequence just in case the prime factorization of $n + 1$ encodes the sequence according to the preceding scheme. Now 0 codes () because $0 + 1 = \Pi_{i \leq 0} p(i)^0$.

Hence 5 encodes $(0,0)$ because $5+1=6=2^{0+1}3^{0+1}$, but so does 12 because $12=2^{0+1}3^{0+1}5^{0}7^{0+1}$ because the "halt sign" (a 0 exponent) occurs in position 2.

There is no question of this coding being primitive recursive, since it is polyadic (it takes arbitrary, finite numbers of arguments) and no primitive recursive function is polyadic. But we can primitive recursively decode such code numbers as follows.

To decode a number n as a sequence, we have to remember to subtract one from each exponent in the prime factorization of $n+1$. The standard notation for the x^{th} position in the sequence coded by the following:

2.10.2 Decoding

$(y)_x =$ the item occurring in the x^{th} position in the sequence coded by y

To keep the function total, we take the first position to be position 0.

$$(y)_x = [y+1]_x \dot{-} 1.$$

2.10.3 Length of decoded sequence

(number of consecutive prime factors of x)
Using the convention that every position in the sequence corresponds to a prime factor, we can simply search for the first non-factor of the code number. Since we count prime factors starting with 0, this corresponds to the length of the coded sequence.

$$lh(x) = \min_{z \leq x} \neg(p(z)|x).$$

Now define n is a **Gödel number** of $\vec{x}$ just in case $\vec{x} = ((n)_0, \ldots, (n)_{lh(n)})$. There will be infinitely many Gödel numbers for each finite sequence, but each such number codes a unique sequence due to the existence and uniqueness of prime factorizations (the fundamental theorem of arithmetic).

It will sometimes be useful to choose the least such Gödel number for a sequence. That is given by

$$\langle x_0, \ldots, x_n \rangle = \Pi_{i \leq n} p(i)^{x_i+1}.$$

Then we have

$$(\langle x_0, \ldots, x_n \rangle)_i = x_i,$$

for each $i \leq n$.

Exercise 2.4 *Bijective Binary Sequence Encoding*
Define the binary primitive recursive function

$$\langle x, y \rangle = \frac{1}{2}[(x+y)(x+y+1)] + x.$$

1. *This coding is evidently primitive recursive. Show that it is a bijection* $\mathbf{N}^2 \mapsto \mathbf{N}$. *Hint: this is just a polynomial expression for the obvious enumeration procedure. Think of the pairs as being presented in an* $\mathbf{N} \times \mathbf{N}$ *array with* $\langle 0, 0\rangle$ *at the upper left. Given* $\langle x, y\rangle$, *recover the code number by counting pairs, starting at* $\langle 0, 0\rangle$, *along diagonals, from the upper right to the lower left.*

2. *Show that*

$$\begin{aligned} \frac{1}{2}\big[(x+y)(x+y+1)\big] &= \sum_{t \leq x+y} t \\ &= \textit{the number of pairs } \langle z, w\rangle \textit{ occuring in} \\ &\quad\; \textit{diagonals to the upper left of } \langle x, y\rangle. \end{aligned}$$

3. *Show that* $x =$ *the number of pairs remaining to be counted to the upper right of* $\langle x, y\rangle$.

4. *Show that the decoding functions are also primitive recursive.*

5. *Use the preceding results and codings to produce* n*-ary primitive recursive encodings and decodings.*

2.11 Fancy Recursion

One reason codings are nice is that they give us new and more elegant forms of recursion for free. The basic idea is that the coding allows primitive recursion to simulate the fancy form of recursion by looking at successive code numbers of the current "computational state" of the fancy form of recursion.

Exercise 2.5 *Course of Values Recursion (Péter)*
Suppose that $s_1(x), \ldots, s_k(x) \leq x$.
Then $CVR(g, f, s_1, \ldots, s_k)$ *is the unique function* h *such that:*

$$\begin{aligned} h(0, y) &= g(y) \\ h(n+1, y) &= f(h(s_1(n), y), \ldots, h(s_k(n), y), n, y). \end{aligned}$$

Use the Gödel coding to show that the set $Prim$ *is closed under the* CVR *operator.*

Exercise 2.6 *Fibonacci Sequence*
Show that the following function is primitive recursive:

$$\begin{aligned} f(0) &= 1; \\ f(1) &= 1; \\ f(x+2) &= f(x) + f(x+1). \end{aligned}$$

Exercise 2.7 *Simultaneous Recursion (Péter)*
$SR_i(g_1, \ldots, g_k, f_1, \ldots, f_k)$ *is the unique function* h_i *such that:*

$$\begin{aligned} h_i(0, y) &= g_i(y); \\ h_i(n+1, y) &= f_i(h_1(n, y), \ldots, h_k(n, y), n, y). \end{aligned}$$

Notice that k *is constant. Use the* k*-ary sequence encoding of exercise 1 to show that* $Prim$ *is closed under the* SR *operator.*

Exercise 2.8 *Recursion with Parameter Substitution*
$PS(g, f, s)$ *is the unique* h *such that:*

$$\begin{aligned} h(0, y) &= g(y); \\ h(n+1, y) &= f(h(n, s(n, y)), n, y). \end{aligned}$$

Show that $Prim$ *is closed under the* PS *operator.*
Hint: Trace the computation of $h(3, y)$*. You will find that you end up needing to know the values of the successive compositions:*

$$\begin{array}{r} y \\ s(2, y) \\ s(1, s(2, y)) \\ s(0, s(1, s(2, y))) \\ \textit{etc.} \end{array}$$

These can be obtained by a "backwards" recursion – or rather a forwards recursion that goes backwards in time!

$$\begin{aligned} S(0, n, y) &= y; \\ S(t+1, n, y) &= s(n \dot{-} t, S(t, n, y)). \end{aligned}$$

Then we have:

$$\begin{aligned} S(0, 3, y) &= y \\ S(1, 3, y) &= s(2, y) \\ S(2, 3, y) &= s(1, s(2, y)) \\ S(3, 3, y) &= s(0, s(1, s(2, y))) \\ \textit{etc.} & \end{aligned}$$

Now write a recursion that applies f *to the right values of* S*.*

Exercise 2.9 *Let* $f(x) = ggg \ldots ghs \ldots sss(x)$*, with* x *iterations of the* g*'s and the* s*'s. Show that* f *is primitive recursive. Hint: use* PS*.*

2.12 Breaking the Primitive Recursion Barrier

We have lots of examples of recursion operations that yield nothing new. This may lull you into assuming that every kind of recursion can be massaged into primitive recursion by means of suitable codings. But that would be a mistake, for there is an intuitively effective recursion operator that generates non-primitive recursive functions.

2.12.1 Double recursion

$R(g_1, g_2, g_3, g_4)$ is the unique h such that:

$$\begin{aligned} h(0, y, z) &= g_1(y, z); \\ h(x+1, 0, z) &= g_2(h(x, c, z), x, z); \\ h(x+1, y+1, z) &= g_4(h(x+1, y, z), h(g_3(h(x+1, y, z), x, y, z), x, z), x, y, z). \end{aligned}$$

Double recursion allows one to apply a variable number of primitive recursions depending on the value of x. To see this, observe that:

- For a given value of x, the third clause decrements y down to 0.
- When y reaches 0, you hit the second clause, which decrements x to $x - 1$ and restarts the primitive recursion with y set to some fixed constant c.
- Finally, when x is decremented down to 0, we hit the first clause, which is the base case.

Since double recursion can simulate arbitrary numbers of primitive recursions, it is fairly intuitive that a double recursive function ought to be able to grow faster than any given primitive recursive function. If the function uses n primitive recursions at stage n, then for each number of primitive recursions, after some time it does something that would require at least one more primitive recursion. The proof of this claim will wait a bit until we introduce the Grzegorczyk hierarchy.

2.12.2 The Ackermann function (1928)

The Ackermann function is historically the first example of an intuitively effective function that is not primitive recursive. (cf. Rogers, p. 8). The Ackermann function grows so fast that some skeptical finitistic mathematicians have denied that it is computable (or that it even exists). This sounds crazy until one considers that the exact values for small arguments can only be defined (in a book length definition) by the function itself! The Ackerman function may be defined as follows:

1. $a(0, 0, z) = z$;
2. $a(0, y+1, z) = a(0, y, z) + 1$;

3. $a(1, 0, z) = 0$;

4. $a(x + 2, 0, z) = 1$;

5. $a(x + 1, y + 1, z) = a(x, a(x + 1, y, z), z)$.

Exercise 2.10 *The Ackermann function is more intuitive than it looks at first. This will be apparent after you do the following.*

1. *Show that the Ackermann function really does result from applying DR to primitive recursive functions. Don't let all those clauses scare you!*

2. *Find simple primitive recursive definitions of each of the functions* $g_i(x, y) = a(i, x, y)$*. Prove that you are correct.*

3. *Relate this fact to the point made about uniformity when we introduced bounded sums as an operator.*

4. *What are* g_0*,* g_1*, and* g_2*?*

2.12.3 The Péter Function

The Péter function is also not primitive recursive. It is a simplified version of Ackermann's function and is called the Ackermann function in Cutland (p. 46).

$$\begin{aligned} p(0, y) &= y + 1 \\ p(x + 1, 0) &= p(x, 1); \\ p(x + 1, y + 1) &= p(x, p(x + 1, y)). \end{aligned}$$

2.13 Even Fancier Recursion

One can generalize the idea of double recursion by adding recursive variables and piling on new recursion conditions. So while double recursion does a variable number of nested primitive recursions, triple recursion allows for a variable number of double recursions, and so forth. Each time we get something new.