The Theory of Computation
Bernard M.E. Moret
Addison-Wesley-Longman, 1998

Solution Manual
Chapter 5, Text Problems

Exercise 5.1

Almost all of these are easy to define informally; the point of the exercise is to write completely formal definitions—i.e., to write programs in the primitive recursive definition language that a compiler would accept as syntactically correct.

1. We use primitive recursion to distinguish the case $n = 0$ from the others:

$$\begin{cases} \mathrm{lev}(0) = 0 \\ \mathrm{lev}(n+1) = \mathrm{Succ}(\mathrm{Zero}(\Pi_1^2(n, \mathrm{lev}(n)))) \end{cases}$$

2. Much the same definition:

$$\begin{cases} \mathrm{lev}(0) = 1 \\ \mathrm{lev}(n+1) = \mathrm{Zero}(\Pi_1^2(n, \mathrm{lev}(n))) \end{cases}$$

3. We use recursion on the second argument (we have seen that we can shift the recursion onto any argument).

$$\begin{cases} \mathrm{minus}(x, 0) = \Pi_1^1(x) \\ \mathrm{minus}(x, n+1) = \mathrm{dec}(\Pi_2^3(x, \mathrm{minus}(x, n), n)) \end{cases}$$

4. This one follows the model of addition, this time using repeated additions to carry out the multiplication..

$$\begin{cases} \mathrm{mult}(0, x) = \mathrm{Zero}(x) \\ \mathrm{mult}(n+1, x) = \mathrm{add}(\Pi_2^3(n, \mathrm{mult}(n, x), x),\ \Pi_3^3(n, \mathrm{mult}(n, x), x)) \end{cases}$$

5. This is the most interesting of the bunch, mostly because of the lazy evaluation. We write $\#(x, y)$ to preserve function notation in the definition, even though we normally use the function in infix notation.

$$\begin{cases} \#(0, y) = \mathrm{Zero}(y) \\ \#(n+1, y) = \Pi_3^3(n, \#(n, y), y) \end{cases}$$

   Note that, since the projection function is lazy, it will not evaluate its first or second argument and thus recursion is not actually invoked; more importantly, the Zero function does not evaluate its argument either, so that the base case functions without examining or evaluating the argument y at all.

Exercise 5.2

We follow the text in defining the function $p(n, x)$, which is intended to return the value $\langle n+1, f(n, x), \ldots, f(), x)\rangle_{n+2}$, using primitive recursion:

$$p(0, x) = \langle 1, g(x)\rangle$$

which is a primitive recursive function of one variable, as desired, and

$$p(n+1, x) = q(n, p(n, x), x)$$

where q is defined through substitution as

$$q(a, b, c) = \langle \mathrm{Succ}(\mathrm{Succ}(\Pi_1^3(a, b, c))), \rangle h(\Pi_1^3(a, b, c), \Pi_3^3(a, b, c), \Pi_2^3(a, b, c)), \Pi_2(\Pi_2^3(a, b, c))\rangle\rangle$$

where each function used is primitive recursive (but where we have slightly simplified the formal definition of q through substitution in order to avoid a long list of intermediate functions).

Now we can formally define f through primitive recursion as

$$\begin{cases} f(0, x) = g(x) \\ f(n+1, x) = h(\Pi_1^3(n, f(n, x), x), \Pi_3^3(n, f(n, x), x), p(\Pi_1^3(n, f(n, x), x), \Pi_3^3(n, f(n, x), x))) \end{cases}$$

where no recursion is used directly—the only use of the primitive recursive mechanism is in distinguishing the case $n = 0$ from the others.

Exercise 5.3

We begin by defining a two-case scheme; multiple cases are an easy generalization from there. We have seen two mechanisms for definition by cases already, both somewhat specialized: one is the primitive recursion scheme, which distinguishes between a base case and a recursive case; and the other is the guard function, which is very similar to our problem, but returns 0 in one of the cases.

A fairly easy solution is to use the guard function twice, once with predicate P and once with predicate $notP$, and concatenate the two results—one of the two will always be 0. This idea yields the definition

$$f(x_1, \ldots, x_n) = \mathrm{con}_2(P(x_1, \ldots, x_n)\#g(x_1, \ldots, x_n), (\mathrm{not}P(x_1, \ldots, x_n))\#f(x_1, \ldots, x_n))$$

which is a valid primitive recursive definition by substitution and clearly does what is intended. It remains to verify that the semantics of the definition allow the function to return one of g or h without computing the other. We have already noted that the guard function, when its predicate is false (when it returns 0), does not evaluate its other argument; thus, when our function evaluates to the value of g, the other argument (the function h) of the second guard is not evaluated, and vice versa.

To generalize to multiple cases, we use concatenation of all of the cases, each guarded with a predicate that excludes all other cases; since the logical connectives are all primitive recursive, we can construct the appropriate predicates; the rest is clearly primitive recursive.

Exercise 5.4

Recursion is the key to these definitions. We begin with the existential quantifier; write it as the function $f(x, z_1, \ldots, z_n)$. We can easily write the base case as

$$f(0, z_1, \ldots, z_n) = P(0, z_1, \ldots, z_n)$$

and can then use the step to write

$$f(n+1, z_1, \ldots, z_n) = f(n, z_1, \ldots, z_n) \text{ or } P(n+1, z_1, \ldots, z_n)$$

describing the fact that the function is true (the bounded existential quantifier succeeds) whenever a value at most n is found that works (the recursive call) or the value $n+1$ itself works.

The bounded universal quantifier is the same, except that it uses a logical *and* rather than a logical *or* to join the result of the recursive call and the evaluation of the predicate on the value $n+1$. Writing $g(x, z_1, \ldots, z_n)$ for this predicate, we get

$$\begin{cases} g(0, z_1, \ldots, z_n) = P(0, z_1, \ldots, z_n) \\ g(n+1, z_1, \ldots, z_n) = g(n, z_1, \ldots, z_n) \text{ and } P(n+1, z_1, \ldots, z_n) \end{cases}$$

Exercise 5.5

- Recursion is the key to this definition. We can easily write the base case as

$$f(0, z_1, \ldots, z_n) = \text{is\_zero}(P(0, z_1, \ldots, z_n))$$

  and can then use the step to write

$$f(n+1, z_1, \ldots, z_n) = \begin{cases} f(n, z_1, \ldots, z_n) & \text{if } f(n, z_1, \ldots, z_n) \leq n \\ n+1 & \text{if } P(n+1, z_1, \ldots, z_n) \\ n+2 & \text{otherwise} \end{cases}$$

  The reader will recall that the ordering of the cases is significant. Here we use recursion to check the smaller values of the first argument; if the returned value is within the range (that is, a solution was found), it is returned as the answer; otherwise the larger value $n+1$ is tried; and if that also fails, then the first value outside the range, namely $n+2$, is returned.
- A simple affair if we remember to think in terms of strings as well as numbers:

$$(x \leq y) = \text{is\_zero}(\text{minus}(x, y))$$

- The key is to use multiplication, which we know to be primitive recursive. Note that x divides y evenly if and only if there exists a z such that $x \cdot z = y$. We need to define the equality predicate first, since we do not have one yet; to avoid confusion, let us denote it by equal(x, y); we have equal$(x, y) = \text{is_zero}(\text{con}_2(\text{minus}(x, y), \text{minus}(y, x)))$. Now we can just write

$$x|y = \exists z/y[\text{mult}(x, z) = y]$$

- Now we use the quantifiers and the division predicate just defined:

$$\text{is\_prime}(x) = \forall y \leq x[2 \leq |y| < |x| \rightarrow \mathit{not}(y|x)]$$

- We take the 0th prime to be 2 (thus prime(0) = 2, prime(1) = 3, prime(2) = 5, etc.) We need a number of helper functions. Basically, to construct the $n+1$st prime, we start from the nth prime and search upwards, testing each successive number for primality. We know that the search need not go beyond some simple function of the

nth prime; trivially, prime$(n+1)$ is at most prime$(n)!+1$, although in fact one can prove that prime$(n+1)$ is at most prime$^2(n)$. We have not seen a primitive recursive definition for the strict inequality, but it is easy to define: $(x < y) = (x \leq y) \wedge not(eq(x, y))$. We shall also need the function to establish the search range; if we use factorial plus one, we can write $fp1(x) = \text{Succ}(fact(x))$, with $fact(0) = 1$ and $fact(n+1) = mult(n+1, fact(n))$, where we omitted the use of the projection functions. Finally, we shall need a search for the smallest element in a range, i.e., a function

$$h(x, z_1, \ldots, z_n) = \min y \leq x P(y, z_1, \ldots, z_n)$$

which returns the smallest y (prefix of x) such that the predicate holds, or returns Succ(x) if no such y exists. This function is much easier to define than it would seem (see also the solution to Exercise 5.10): we set

$$h(0, z_1, \ldots, z_n) = lev'(P(0, z_1, \ldots, z_n))$$

and

$$h(x+1, z_1, \ldots, z_n) = \begin{cases} h(x, z_1, \ldots, z_n) & \text{if } h(x, z_1, \ldots, z_n) \leq x \\ \text{Succ}(x) & \text{if } P(\text{Succ}(x), z_1, \ldots, z_n) \\ \text{Succ}(\text{Succ}(x)) & \text{otherwise} \end{cases}$$

Now we can write prime(x) as

$$\begin{cases} \text{prime}(0) = 2 \\ \text{prime}(n+1) = \min y \leq fp1(\text{prime}(n))[(y > \text{prime}(n) \wedge \text{is\_prime}(y)] \end{cases}$$

Exercise 5.6

How to obtain a unique code for a primitive recursive function is clear; the only trick is to add to the code the number of arguments of the function (for reasons that will become clear on verifying the code); the steps thus are

- Zero( ) is coded $\langle 1, 0\rangle_2$.
- Succ( ) is coded $\langle 1, 1\rangle_2$.
- $P_i^j(\,)$ is coded $\langle j, 2, i\rangle_3$.
- If f is a function of n arguments defined by substitution from the functions g (of m arguments) and $\{h_i \mid 1 \leq i \leq m\}$, where g has code c_g and h_i has code c_i for each i, then f is coded by $\langle n, 3, m, c_g, c_1, c_2, \ldots, c_m\rangle_{m+3}$.

- If f is a function of n arguments defined by primitive recursion from the functions g (of $n-1$ arguments) and h (of $n+1$ arguments), where g has code c_g and h has code c_h, then f is coded by $\langle n, 4, c_g, c_h \rangle_3$. (This ignores the small problem when n is 1, which we can easily handle by a separate case.)

Now the enumeration proceeds simply as follows

```
i := 1
repeat
  if is_valid_code(i,Pi1(i)) % # of arguments is unknown
    then print decode(i)
  i := i+1
forever
```

In order for this program to work, we need to define `is_valid_code( )` and `decode( )`. For a code to be valid, we must check that (i) the outermost definition is valid (one of the five above); and (ii) if the outermost definition is of type 3 or 4, the codes of the functions used in the definition are themselves valid. Thus we get something like this (in some convenient pseudocode):

```
is_valid_code(x,n) ==
  % x is the code and n is the expected number of arguments
  if Pi1(x) <> n  % does the # of arguments match expectations?
    then
      false
    else set code Pi2(x) % for clarity
      if (code = 0) or (code = 1))  % Zero() or Succ()
        then true
      else if (Pi1(code) = 2)   % projection, check 1<=i<=j
        then (1 <= Pi2(code) <= Pi1(x))
      else if (Pi1(code) = 3)   % substitution, check pieces
        then set m Pi1(Pi2(code))  % for clarity
             (is_valid_code(Pi_2^(m+1)(Pi2(code)),m) and
              for all i, 1 <= i <= m,
                is_valid_code(Pi_(i+2)^(m+1)(Pi2(code)),n))
      else if (Pi1(code) = 4)   % prim. recursion, check pieces
        then (is_valid_code(Pi1(Pi2(code)),n-1) and
              is_valid_code(Pi2(Pi2(code)),n+1))
        % this again ignores the case n=1
      else false  % not one of the 5 recognized patterns
```

The initial call to the checking routine does not actually check arguments, since it uses as expected number of arguments the number coded in—so that the first matching check is guaranteed to succeed. After that, however, it is critical that the number of arguments be verified at each step. Now the function `decode()` is similar, but in some ways simpler, since it no longer needs to check the syntax, but can assume that the definition is valid.

```
decode(x) ==
  set code Pi2(x)
  print("Code ",code," defines a function of ",Pi1(x)," arguments.")
  case code of
    0: print("The function is Zero( )")
    1: print("The function is Succ( )")
    2: print("The function is the projection of ",Pi2(code))
    3: print("The function is defined by substitution;
       print("the outer function is as follows:")
       decode(Pi_2^(m+1)(Pi2(code)))
       print("and the inner functions are as follows:")
       for i:=1 to Pi1(Pi2(code)) do
          decode(Pi_(i+2)^(m+1)(Pi2(code)))
    4: print("The function is defined by primitive recursion;")
       print("the base function is as follows:")
       decode(Pi1(Pi2(code)))
       print("the recursive step function is as follows:")
       decode(Pi2(Pi2(code)))
  print("End of definition of code ",code)
```

Exercise 5.7

This is actually very simple: all we need is an external induction, much like our definition of the family of concatenation functions. We have $f_0(x) = \mathrm{Succ}(x)$, which is primitive recursive and $f_1(\epsilon, x) = P_1^1(x)$ and $f_1(ya, x) = \mathrm{Succ}(P_2^3(y, f_1(y, x), x))$, which is also primitive recursive. Now let us assume that all $f_i(y, x)$ are primitive recursive up to some $i = k$ and let us prove that $f_{k+1}(y, x)$ is also primitive recursive. By induction, it will follow that all f_i in the family are primitive recursive. We have, by definition, $f_{k+1}(\epsilon, x) = P_1^1(x)$, which obeys all requirements for the basis of primitive recursion, and $f_{k+1}(ya, x) = f_k(P_2^3(y, f_{k+1}(y, x), x))$, which obeys all requirements for the step in primitive recursion. Since all functions used in the definition are primitive recursive (trivially or by inductive hypothesis) and since the definition is a valid application of primitive recursion, the resulting function is primitive recursive.

Exercise 5.8

The reason is that the definition of the new "diagonal" function, $g(n) = f_n(n) + 1$, is not a primitive recursive definition—it cannot always be defined with just substitution and primitive recursion. We saw a detailed example when looking at Ackerman's function, which uses much the same definitional scheme, namely, "use the nth function on the nth argument," and showed then that the function thus defined was not primitive recursive.

The key to a contradiction proof by diagonalization always resides in constructing a new object that is not in the set (by virtue of its construction), but which in fact ought to be in the set if the claimed negated result were to be true, yielding the desired result through contradiction. The proposed construction does yield a new total function, but we have no reason to believe it to be in the set of primitive recursive functions; on the other hand, the very same argument applied to all computable total functions works, because we have very few restrictions left when defining arbitrary (partial or not) recursive functions and thus the new function g should now be in the set if computable total functions.

Exercise 5.9

This is not really a problem, but rather a reason to go back through the definition of the various functions used in the arithmetization of Turing machines, so as to verify that all of the functions that appear in the definition of the composition of two Turing machines are in fact primitive recursive functions. Since in fact every single function defined throughout the arithmetization process is primitive recursive, with the single exception of the function stop (and of its three derivatives, the functions θ, out, and ϕ_{univ}), as the reader is invited to verify once again, the result follows.

Exercise 5.10

There are several ways to attack this problem. Perhaps the simplest way is to use the step function that was defined for Turing machines.

The $\{\phi_i\}$ programming system that we developed to simulate a Turning machine already has a step function that is very close to what we need. We need only modify it to make it start from the initial state and to check whether the Turing machine has reached the halt state after m steps; if it has, we clean up the tape as described in the text and return the output value (plus 1), otherwise we return ε. We need to translate back and forth, which requires universal and s-1-1 functions on each side—which we have, since we are working with acceptable programming systems. So, to get step(i, x, m) in our $\{\psi_j\}$ system, we translate i into the $\{\phi_i\}$ system, obtaining say $t(i)$, then define our modified Turing machine step function in terms of $t(i)$, x, and m (where m is taken to be a number of steps) to obtain $\text{step}_{TM}(t(i), x, m)$, which is some (total) function ϕ_k in the TM system; we then translate back from ϕ_k to the $\{\psi_j\}$ system, obtaining some (total) function $\psi_{t'(k)}$. Now, by construction, we have $\psi_{t'(k)}(< t(i), x, m >) = \text{step}_{TM}(t(i), x, m)$, which returns either ε or $\text{Succ}(\phi_{t(i)}(x)) = \text{Succ}(\psi_i(x))$. It is easily verified that, whenever $\psi_i(x)$ is defined, there must exist some value m_0 of m (the number of steps required by TM $t(i)$ to stop on input x) such that $\psi_{t'(k)}(< t(i), x, m_0 >) = \text{Succ}(\psi_i(x))$, whereas otherwise

$\psi_{t'(k)}(< t(i), x, m >) = \varepsilon$ for all m. Now we write formally step$(< i, x, m >) = \psi_{c(t'(k),l}(< i, x, m >)$, where c is the composition function and l is the index of the total recursive function $\psi_l(z) =< t(\Pi_1(z)), \Pi_2(z) >$.

In a second approach, we can just take the Turing machine program that implements the step function and encode it directly into our $\{\psi_j\}$ system rather than consider its encoding into the $\{\phi_i\}$ system and translate back and forth.

A more ambitious approach remains purely internal to our programming system. Suppose that, for each fixed m, we defined a new, bounded version of μ-recursion as follows: given function f of $n + 1$ arguments, we define the function g of n arguments by m-bounded μ-recursion as the successor of the smallest z, $z \leq m$, such that $f(z, x_1, \ldots, x_n) = \epsilon$; if no z is found that meets the criteria, g returns ϵ. Because the search on z is bounded by m, it always terminates. Thus, provided that f is total, so is g. Any number of applications of m-bounded μ-recursion construct only total functions. If the original (unbounded) μ-recursion terminates on arguments $x_1, \ldots, x_n$, then there must exist an m (the value returned in the original μ-recursion, in fact) such that the m-bounded recursion finds a z_0 meeting the predicate and thus returns Succ(z_0), whereas, if the original μ-recursion would have produced an undefined result, then the m-bounded μ-recursion always returns ε. If we replace μ-recursion by m-bounded μ-recursion in the definition of ψ_j in our system, then correct for the use of *Succ* and the ε flag, to propagate ε throughout and to remove the Succ as needed, we shall obtain a new recursive function, call it f_j^m. Whenever $f_j^m(x) \neq \varepsilon$, then $\psi_j(x)$ is defined and $f_j^m(x) = \text{Succ}(\psi_j(x))$. Note that everything remains properly defined within our original programming system, simply because bounded μ-recursion is not really a new definitional scheme, but a higher-level composition operator that can be defined in terms of the original primitive recursive base functions and schemes.

So now we need only show that there is a single recursive function step$(< i, x, m >)$ such that step$(< i, x, m >) = f_j^m(x)$ and we will be done. But recall that the index i is the coding of the sequence of applications of the three construction schemes to the base functions that define the function ϕ_i; we can decode this index i to obtain the various pieces and replace each application of μ-recursion by an application of m-bounded μ-recursion (which has its own code, since we can define it through primitive recursion), then we can recode the complete index to obtain the index of f_j^m within $\{\psi_j\}$; finally, we can invoke ψ_{univ} to execute the function given by this index.

Exercise 5.11

1. Note that this function is only defined for $n > 0$: 0 itself does not have a prime power decomposition. We know from Exercise 5.5 that the function prime(i), which returns the ith prime, is primitive recursive, as is the bounded maximization operator (see below), as is the predicate $x|y$. We can define the primitive recursive function power(i, n), which raises n to the ith power, as follows:

$$\begin{cases} \text{power}(0, n) = 1 \\ \text{power}(i + 1, n) = \text{mult}(n, \text{power}(i, n)) \end{cases}$$

(where we omitted the projection functions). An easy definition of exp(n, m) can then be had by substitution:

$$\text{exp}(n, m) = \max y \leq n[\text{power}(y, \text{prime}(m))|n]$$

2. We can define this function by primitive recursion; call the function $f(x, z_1, \ldots, z_n)$ and define:

$$\begin{cases} f(0, z_1, \ldots, z_n) = g(0, z_1, \ldots, z_n) \\ f(n + 1, z_1, \ldots, z_n) = \text{larger}(f(n, z_1, \ldots, z_n), g(n + 1, z_1, \ldots, z_n)) \end{cases}$$

(omitting the projections needed to make this perfectly legal), where larger is defined by substitution as

$$\text{larger}(x, y) = \text{con}_2(\text{lev}(\text{minus}(x, y))\#x, \text{is\_zero}(\text{minus}(x, y))\#y)$$

using much the same idea (concatenation of two arguments, one of which is always 0) as in the definition by cases of Exercise 5.3.

3. With the course-of-values recursion of Equation 5.1 (and Exercise 5.2), this problem is a breeze! The general definition just uses Equation 5.1 to write:

$$f(n+1) = \text{add}\,(\Pi(2, n+1, \langle n+1, f(n), f(n-1), \ldots, f(0)\rangle), \Pi(3, n+1, \langle n+1, f(n), f(n-1), \ldots, f(0)\rangle))$$

That is, the function h of the definition is a function that adds the second and the third terms in the pairing and returns that sum. We do need to specialize the definition of Equation 5.1 to a function f of just one variable, so that the base case invokes a function g of zero variables—a constant. However, it is not quite enough to write

$$f(0) = 1$$

since we need two base cases, not just one. But we can handle this problem rather easily through a definition by cases and an artificial shift of arguments.

We can now give the correct definition. We define f by cases:

$$f(n) = \begin{cases} 1 & n = 0 \\ f'(n-1) & \text{otherwise} \end{cases}$$

and define f' by course-of-values recursion, where the key function p is slightly modified to p', yielding

$$\begin{cases} f'(0) = 1 \\ f'(n+1) = h(n, p'(n)) \end{cases}$$

The modified p is defined by

$$\begin{cases} p'(0) = \langle 1, 1 \rangle \\ p'(n+1) = \langle h(n, p'(n)), p'(n) \rangle \end{cases}$$

so as to include the two base cases at once—and also simplified by omitting the index in the pairing (since we always know that value of that index). Finally, as we mentioned above, the h adds the now first and second (because we dropped the index) terms of its pairing, so we define it as

$$h(i, j) = \text{add}(\Pi(1, i+2, j), \Pi(2, i+2, j))$$

(where the $i + 2$ is due to the initial pairing of two base values).

This technique shows us how to handle the definition of recurrences of any constant order: just pack all the base cases into one pairing as the base case of a modified p, then define the main function by cases and by shifting the argument.

Exercise 5.12

We are given a primitive recursive function $g(y)$. Then define

$$\begin{cases} f(0, y) = \Pi_1^1(y) \\ f(n+1, y) = g(\Pi_2^3(n, f(n, y), y)) \end{cases}$$

which define $f(n+1, y)$ to be $g(f(n, y))$ and thus uses recursion to obtain the iteration. We have basically given a tail-recursive mechanism for iteration, as is done in any functional programming language.

Exercise 5.13

We are given primitive recursive functions g and h. Note that the effect of the definitional mechanism given is to define $f(n, x)$ to be $g(h(h(\ldots h(x) \ldots)))$, with n nestings of h. We can obtain this result by using iteration as in Exercise 5.12 and ending with one application of g. Thus we define

$$f(n, x) = g(f'(n, x))$$

and now use iteration to define $f'(n, x)$ to be $h^n(x)$:

$$\begin{cases} f'(0, x) = x \\ f'(n+1, x) = h(f'(n, x)) \end{cases}$$

Exercise 5.14

We do not give any solution here, since the choice of language and environment so strongly affects the outcome. We simply point out that it is crucial to consider the range of integers supported by the type used in the functions, since otherwise overflow is sure to occur very early on. The alternate definition of Ackerman's function given on p. 132 (using double recursion) can also be used, although using the completion version is better preparation for the second part of the problem. We also point out that a functional language (Lisp, Scheme, etc.) is particularly well-suited to the task, since the definition of a function sitting much higher in the hierarchy (say the completion of the second level) requires a highly recursive definition in which arguments of the recursive definition are themselves functions.

Exercise 5.15

1. Define the function $g(<x, y>) = \phi_{\text{univ}}(x, x) - \phi_{\text{univ}}(x, x)$; because this function is partial recursive, there exists some index j such that $\phi_j(<x, y>) = g(x, y)$. By the s-1-1 theorem, there exists an index $k = s(j, x)$ such that $\phi_k(y) = \phi_j(<x, y>)$ for each fixed x. But note that $\phi_k(y) = \phi_j(<x, y>) = g(<x, y>) = \phi_{\text{univ}}(x, x) - \phi_{\text{univ}}(x, x)$ is totally undefined whenever $x \notin K$ and is the constant function ϵ whenever $x \in K$. Thus k is the index of a constant function if and only if $x \in K$; since K is not recursive, neither is the set of constant functions.
2. The very same definition works: the function we obtained is totally undefined if and only if x is not in K, the desired reduction. (A constant function is just a special case of a not totally undefined function.)

3. Define the function $g(<x, y>) = y + \phi_{\text{univ}}(x, x) - \phi_{\text{univ}}(x, x)$; because this function is partial recursive, there exists some index j such that $\phi_j(<x, y>) = g(x, y)$. By the s-1-1 theorem, there exists an index $k = s(j, x)$ such that $\phi_k(y) = \phi_j(<x, y>)$ for each fixed x. But note that $\phi_k(y) = \phi_j(<x, y>) = g(<x, y>) = y + \phi_{\text{univ}}(x, x) - \phi_{\text{univ}}(x, x)$ is totally undefined whenever x is not in K and returns its argument y otherwise. Thus k is not in our set whenever x is not in K and it is in our set (because the range of the function is then the whole of $\mathbb{N}$ and thus encompasses the indices of all total functions and thus of at least one total function) whenever x is in K.

Exercise 5.16

1. $S(y)$ is r.e., but not recursive; in consequence, its complement is not r.e. Rice's theorem applies directly to $S(y)$: the set is characterized by input-output behavior and is neither empty nor the universe; thus $S(y)$ is not recursive. Intuitively, $S(y)$ is r.e. because we can try all functions for all arguments for all steps, printing the function index whenever that function stops for that argument and that number of steps and returns y. Formally, for each fixed y, we can define a function

$$\theta(x) = \mu z[\text{step}(x, \Pi_1(z), \Pi_2(z)) = \text{Succ}(y)]$$

   For a given value of x, our function converges if and only if, for some argument and some number of steps, $\theta(x)$ returned y; thus the domain of θ is exactly $S(y)$.

2. By Rice's theorem, this set is not recursive: the property is a property of functions, not a property of programs, and is non-trivial (there are both injective and non-injective functions). It follows by closure that the complement is also non-recursive. We need to be a bit more specific about injectivity before we decide whether or not the set or its complement is r.e.; the definition of injectivity normally given is $\forall x, y, \ (x \neq y) \Longrightarrow (f(x) \neq f(y))$, which does not carry over well to non-total functions. We can simply ignore all arguments on which the function is not defined and state that ϕ is injective if, whenever $\phi(x)\downarrow$ and $\phi(y)\downarrow$, then $(x \neq y) \Longrightarrow (\phi(x) \neq \phi(y))$, or we can state that only total functions can be injective. We look at both interpretations below. Under the first interpretation (only defined values count), we claim that the complement is r.e., so that the set itself cannot be r.e. To see that the complement is r.e., note that placing some program ϕ_i in this complement set can be done just by identifying two arguments x and y such that $x \neq y$ and yet $\phi_i(x) = \phi_i(y)$. We can do this by quintuple

dovetailing, on all programs, all pairs of arguments, and all steps for each argument; whenever we find a z such that

- $\mathrm{step}(\Pi_1^5(z), \Pi_2^5(z), \Pi_3^5(z)) \neq 0$ and
- $\mathrm{step}(\Pi_1^5(z), \Pi_4^5(z), \Pi_5^5(z)) \neq 0$ and
- $\Pi_2^5(z) \neq \Pi_4^5(z)$ and
- $\mathrm{step}(\Pi_1^5(z), \Pi_2^5(z), \Pi_3^5(z)) = \mathrm{step}(\Pi_1^5(z), \Pi_4^5(z), \Pi_5^5(z))$,

we print $\Pi_1^5(z)$.

Under the second interpretation, however, neither the set nor its complement is r.e. Thus we must do some extra work. We reduce $\overline{K}$ in turn to the set and to its complement. To reduce $\overline{K}$ to the set of (total) injective functions, we define

$$\theta(x, y) = \begin{cases} y & \mathrm{step}(x, x, y) = 0 \\ \phi_j(y) & \text{otherwise} \end{cases}$$

where ϕ_j is the totally undefined function. This new function is clearly partial recursive (a valid definition by cases), so we can write $\theta(x, y) = \phi_i(x, y)$ for some index i. Using the s-m-n construction, we now get $\phi_{s(i,x)}(y) = \theta(x, y)$. But note that $\phi s(i, x)$ is the identity function whenever $x \in \overline{K}$ and is undefined on at least one value of y otherwise. Hence $s(i, x)$ is injective whenever $x \in \overline{K}$ and is not injective (because it is partial) otherwise. Thus we have reduced $\overline{K}$ to our set of injective functions, thereby showing that our set is not r.e. It remains to show that the complement of our set is not r.e. either; for that we need to reduce $\overline{K}$ directly to the complement. We define a new function $\zeta(x, y) = y + \mathrm{Zero}(\phi_{\mathrm{univ}}(x, x))$. Again, this is a partial recursive function, so that there exists an index k such that $\phi_k(x, y) = \zeta(x, y)$; again, the s-m-n theorem tells us that we can write $\phi_{s(k,x)}(y) = \phi_k(x, y) = \zeta(x, y)$. But note that $\phi_{s(k,x)}(y)$ is totally undefined when $x \in \overline{K}$ and is the identity function otherwise; that is, $s(k, x)$ is in the complement of our set whenever $x \in \overline{K}$, so that we have a reduction from $\overline{K}$ to the complement of our set, thereby proving it to be non-r.e.

3. This is a rather tricky question. It depends on what is understood by being primitive recursive. If what is meant is "the program was defined using only the three base functions and the operations of substitution and primitive recursion," then the set is certainly r.e. and in fact recursive—we defined a total recursive predicate in a previous homework. In that case, of course, the complement of the set is also recursive.

However, if what is meant is "the program computes a function that can be implemented by a primitive recursive program," then we have an I/O property (the set of all programs that compute the same function as any of the primitive recursive programs) and thus our set is not recursive by Rice's theorem. Rather worse, our set is then not even r.e., intuitively because it would require us to decide of two programs whether they compute the same function (one being the program in hand and the other some program defined in a primitive recursive fashion). Consider the simple function

$$\theta(x, y) = \begin{cases} 0 & \text{step}(x, x, y) = 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

and use the s-m-n theorem to obtain $\phi_{s(x)}(y) = \theta(x, y)$. Note that $\phi_{s(x)}(y)$ need not be a primitive recursive function as defined. However, if $x \in \overline{K}$, then $\phi_{s(x)}(y)$ is the constant function 0, which is a primitive recursive function (defined by $\text{Zero}(y)$), whereas, if $x \in K$, then $\phi_{s(x)}(y)$ is mostly undefined and thus not primitive recursive. Thus we have reduced $\overline{K}$ to the set of primitive recursive functions.

The complement of the set (the set of non-primitive recursive functions) is equally non-r.e. We prove it by reduction from $\overline{K}$. Define $\theta(x, y) = \text{Zero}(\phi_{\text{univ}}(x, x))$ and use the s-m-n theorem to obtain $\phi_{s(x)}(y) = \theta(x, y)$. If $x \in \overline{K}$, then $\phi_{s(x)}(y)$ is the totally undefined function, which is not primitive recursive and thus is in the complement, while, if $x \in K$, $\phi_{s(x)}(y)$ is the constant function 0, which is primitive recursive.

4. This is an I/O property—it concerns the values returned by the program for increasingly larger arguments; thus the set is not recursive by Rice's theorem. Intuitively, the set is non-r.e., because verifying a growth rate requires verification for all possible arguments. To be more precise, we need to define carefully what is meant by "grow as fast as". If we mean that the function should be at least as large as n^2 everywhere, then the set is clearly non-r.e., since it requires first that the function be total (an easy reduction from $\overline{K}$ is to use the step function mechanism and define the new function to be n^3 when $\text{step}(x, x, xy) = 0$).

   The complement of the set (the functions that on at least one value are smaller than n^2 or are undefined) is also non-r.e., simply because the totally undefined function is part of that set, so that a trivial reduction of the type

$$\theta(x, y) = n^3 + \text{Zero}(\phi_{\text{univ}}(x, x))$$

will work—if $x \in \overline{K}$, then the function is totally undefined and thus on the complement, whereas, if $x \in K$, the function is just n^3, which grows faster than n^2 and thus is not in the complement.

5. Recall that an r.e. set is the range (or domain) of a partial recursive function. Thus our set could equally well have been defined as the set of all partial recursive functions that have at least three elements in their range (or domain). Since having a range (or domain) of size at least three is an I/O property, this set is not recursive by Rice's theorem. The set is r.e., however: intuitively, we can dovetail on all functions, all arguments, and all numbers of steps, keeping track of what converged and printing out a function index as soon as it has converged on at least three distinct arguments (for domain definitions, but range definitions are similar). Formalizing the notion is not hard (we basically want to run step$(\Pi_1^3(z), \Pi_2^3(z), \Pi_3^3(z))$, recording $\Pi_1^3(z)$ and one of $\Pi_2^3(z)$ (for domains) or step$(\Pi_1^3(z), \Pi_2^3(z), \Pi_3^3(z))$ (for ranges) for which the function is non-zero, and printing $\Pi_1^3(z)$ as soon as three distinct values of $\Pi_2^3(z)$ (for domains) or of step$(\Pi_1^3(z), \Pi_2^3(z), \Pi_3^3(z))$ (for ranges) are found that pair with the same $\Pi_1^3(z)$ value.

   Since the set is non-recursive and r.e., its complement is not r.e. This is an r.e. set. Informally, since we can enumerate all Turing machines and enumerate all possible input strings, we can start a triple dovetailing process, on all machines, all inputs, and all steps. In the process, we every now and then discover a pair (machine,input) such that the machine accepts the string; we then "chalk one up" for that machine. As soon as we have chalked up three for a machine, we print it (its index). In that way, we correctly enumerate all machines that accept at least three strings.

6. Neither this set nor its complement is r.e. That neither is recursive is an immediate consequence of Rice's theorem, since being defined on a finite or infinite number of inputs is an I/O behavior. To show that neither S nor $\overline{S}$ is r.e., we reduce $\overline{K}$ to each in turn.

   To reduce $\overline{K}$ to S, we define the function $\theta(x, y) = \phi_{\text{univ}}(x, x)$; since this is a valid partial recursive function, it has some index ϕ_i and we can write $\phi_{s(x)}(y) = \phi_i(x, y) = \theta(x, y)$. Now, if $x \in \overline{K}$, then $\phi_{s(x)}(y)$ diverges for all y, and thus has empty (and hence finite) domain, and thus $s(x)$ belongs to S. On the other hand, if $x \in K$, then $\phi_{s(x)}(y) = c$,

where $c = \phi_x(x)$ is a constant, so that $\phi_{s(x)}$ is total and thus has infinite domain, and hence $s(x) \notin S$, as desired.
To reduce $\overline{K}$ to $\overline{S}$, we define the function

$$\theta(x, y) = \begin{cases} \text{Zero}(y) & \text{step}(x, x, y) = 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

since this is a valid partial recursive function, it has some index ϕ_i and we can write $\phi_{s(x)}(y) = \phi_i(x, y) = \theta(x, y)$. Now, if $x \in \overline{K}$, then $\phi_{s(x)}(y) = \text{Zero}(y)$ and thus is total and has infinite domain, and thus $s(x)$ belongs to $\overline{S}$. On the other hand, if $x \in K$, then $\phi_{s(x)}(y)$ is undefined for all $y \geq y_0$ for some constant y_0, and thus has finite domain and therefore $s(x)$ does not belong to $\overline{S}$, as desired.

7. Exercise 5.15 gave us that all three sets are non-recursive.
   - The first set is the set of constant functions; a standard reduction according to Table 5.1 works directly, showing that this set is non-r.e.—define

     $$\theta(x, y) = \begin{cases} \text{Zero}(y) & \text{step}(x, x, y) = 0 \\ y & \text{otherwise} \end{cases}$$

     Then the resulting function is the constant 0 whenever x belongs to $\overline{K}$ and is a non-constant function otherwise, giving us the desired reduction. The complement of the set is the set of all non-constant functions; again, a direct reduction from $\overline{K}$ works easily—define

     $$\theta(x, y) = \text{Zero}(\phi_{\text{univ}}(x, x))$$

     Then the resulting function is the constant function 0 whenever x does not belong to $\overline{K}$ and the non-constant totally undefined function otherwise.
   - The second set is the set of functions with non-empty domain. This set is r.e.: intuitively, we can dovetail on all programs, all arguments, and all steps and output the index of a program as soon as it is observed to stop on an argument. The complement is thus automatically non-r.e.
   - The third set is the set of functions defined on at least one argument which happens to be the index of a total program. This set is non-r.e. (intuitively, we cannot verify that an index on which the function is defined characterizes a total program).

To prove that this set is non-r.e., we reduce the set H of total functions to it (a much easier reduction than reducing $\overline{K}$). Define

$$\theta(x, y) = \begin{cases} 0 & x = y \\ \text{undefined} & \text{otherwise} \end{cases}$$

Use the s-1-1 theorem to obtain a function $\phi_{g(x)}(y)$; note that this function is defined only at $y = x$. Thus $g(x)$ belongs to our set if and only if $y = x$ (its only convergence value) is the index of a total function, that is if and only if x belongs to H—the desired reduction.
The set's complement is the set

$$\{x \mid \forall y(\phi_x(y) \uparrow \text{ or } \phi_y \text{ is not total})\}$$

It is also a non-r.e. set (intuitively, we cannot verify the "for all" condition); note that it contains the totally undefined function. A simple reduction from $\overline{K}$ proves the case—define

$$\theta(x, y) = y + \text{Zero}(\phi_{\text{univ}}(x, x))$$

The resulting function is totally undefined whenever x belongs to $\overline{K}$ and is the function y otherwise—and that function converges everywhere and so in particular converges on the index of a total program.

Exercise 5.17

If we consider the original busy beaver problem—how long can a program of length n run and still stop, that is clearly undecidable, since otherwise we could easily solve the halting problem: given a program of length n and input y, feed $n + |y|$ to the busy beaver function, get a maximum number of steps and now run the program for at most these many steps on input y: if the program halts, we are done; if it does not, we can legitimately assert that it will not ever halt on y, since we gave it the largest number of steps it could ever require.

The version of the busy beaver problem formulated in terms of output value is equally undecidable: if we could decide it, we could solve a slight variant of the problem of finding the shortest program that prints n and halts (a problem that we showed in the text to be undecidable. The variant simply asks for the length of the shortest program that prints *at least n* and halts—it is clear that our proof in the text applies unchanged to this variant. Our reduction works like this: Given n, the (lower bound on the) number to print, we run a loop on some index i from 1 until success and ask the busy beaver function whether a program of length i can print a number at least as large as n and halt. The search must stop, since each longer program can print a larger number; thus this loop uses a putative solution to the busy beaver problem to provide a solution to the shortest program problem; since finding the shortest program to print (at least) n is undecidable, finding the largest value printed by a program of length n is also undecidable.

We could also use the same strategy we used in proving the shortest program undecidable: a direct contradiction obtained by constructing a new program from the existing one. If we had a routine f that printed the value $f(n)$, the largest number that a program of length n with no input can print and still halt, we could construct the new program g for each fixed m by running the following loop

```
i := 1;
while f(i) < m do i := i+1;
print f(i)
```

This new program prints the smallest value $f(i)$ such that no program of length less than m can print it. Yet our new program g has length $O(\log m)$ and prints $f(i)$, a contradiction, since m grows asymptotically faster than $\log m$. Thus the subroutine f does not exist.

Exercise 5.18

If the set S is empty, the results are trivially true; so let us assume that the set S is non-empty and let f be some recursive function whose range is the set S (one such function must exist by definition of r.e. sets).

We begin with the ranges. and define the new function

$$\theta(z) = \phi_{\text{univ}}(f(\Pi_1(z)), \Pi_2(z))$$

That is, $\theta(\langle x, y\rangle)$ runs the xth program in S (namely $\phi_{f(x)}$) on argument y; the result is, by definition, in the range of $\phi_{f(x)}$ and thus in R. Thus $ran\theta \subseteq R$. Conversely, note that any element $x \in R$ is in $ran\phi_i$ for some $i \in S$; in particular, there exists some y such that $\phi_i(y) = x$. Since f enumerates S, there exists some j such that $f(j) = i$; but then $\theta(\langle j, y\rangle) = \phi_{f(j)}(y) = \phi_i(y) = x$, so that $x \in ran\theta$ and thus $R \subseteq ran\theta$.

Now we move to the domains. What we need to do here is to define a new function ζ that will be defined on any value on which one of the ϕ_x, $x \in S$, is defined. We cannot afford to run the ϕ_x on a given value directly, because it might be undefined and cause our ζ to become undefined; thus we use our old trick of dovetailing with a step function:

$$\zeta(x) = \mu z[\text{step}(f(\Pi_1(z)), x, \Pi_2(z)) > 0]$$

If $x \in dom\zeta$, then there exists $z = \langle i, j\rangle$ such that $\text{step}(f(i), x, j) > 0$, i.e., such that $\phi_{f(i)}(x)\downarrow$, i.e., such that $x \in dom\phi_{f(i)}$, i.e., such that $x \in D$; hence $dom\zeta \subseteq D$. Conversely, if $x \in D$, then there exists $k \in S$ such that $x \in dom\phi_k$; hence there must exist some number of steps, say t, sufficient to ensure that ϕ_k converges on x, i.e., such that $\text{step}(k, x, t) > 0$. But f enumerates S, so that there must exist y such that $f(y) = k$; but then we have $\zeta(x) \leq \langle y, t\rangle$, so that $\zeta(x)\downarrow$, so that $x \in dom\zeta$; hence $D \subseteq dom\zeta$, as desired.

Exercise 5.19

- That is clear from the definition. To be formal, we define the characteristic function of K_t:

  $$c_{K_t}(x) = \begin{cases} 1 & \exists y \leq t\ [\text{step}(x, x, y) > 0] \\ 0 & \text{otherwise} \end{cases}$$

  (or just $c_{K_t}(x) = \text{lev}(\text{step}(x, x, t))$, under the usual assumption that, once a function has converged after some n_0 steps, it also converges for all $n \geq n_0$ steps). Since the predicate is recursive (it is even primitive recursive if we have a primitive recursive step function), we obtain a valid recursive characteristic function. The set K_t is a "sanitized," step-bounded version of K. In particular, note that $K_t \subseteq K_{t+1} \subset K$, with $\lim_{t\longrightarrow\infty} K_t = K$, so that

  $$\bigcup_{t\in\mathbb{N}} K_t = K$$

  (which incidentally fits the result of Problem 2, since all sets involved are r.e.).

- We use the family of K_t sets to provide a counterexample. Note that $\overline{K} \subset \overline{K}_{t+1} \subseteq \overline{K}_t$, with $\lim_{t\longrightarrow\infty} \overline{K}_t = \overline{K}$, so that

  $$\bigcap_{t\in\mathbb{N}} \overline{K_t} = \overline{K}$$

  Since $\mathbb{N}$ and the K_ts are recursive and thus r.e., this is an example of an intersection of r.e. sets indexed by an r.e. set that does not produce an r.e. set. However, the intersection is not precisely of the form given in our problem, since $\overline{K_t}$ is unlikely to equal $dom\phi_t$. We can make the idea work, however, by noting that each $\overline{K_t}$ is recursive, and thus r.e., and thus the domain of some partial recursive function, call it $\phi_{g(t)}$. Now let $S = g(\mathbb{N})$; S is clearly an r.e. set if g is a recursive function. Then

$$\bigcap_{x \in S} dom\phi_x = \bigcap_{t \in \mathbb{N}} dom\phi_{g(t)} = \bigcap_{t \in \mathbb{N}} \overline{K_t} = \overline{K}$$

and is not r.e. Thus it remains only to show that g is a total recursive function; define

$$\theta(t, x) = \begin{cases} 1 & \exists y \leq t \text{ [step}(x, x, y) > 0] \\ \text{undefined} & \text{otherwise} \end{cases}$$

Use the s-m-n theorem to obtain $\phi_{s(t)}(x) = \theta(t, x)$. Note that $dom\phi_{s(t)} = K_t$ by definition; hence $s(t)$ is our desired g.

Exercise 5.20

Intuitively, we can run inside our non-repeating enumerator an enumerating machine that may cause repetitions, but store every string printed so far on a tape and verify that newly generated strings do not appear on the tape before printing them. That is a simple filtering operation; it clearly terminates in finite time (although it is clearly very slow and very space-consuming). If our enumerator does not take any input and runs forever, it is a trivial matter to convert it to a machine that prints the ith element and halts by running the enumerator internally until it has generated i elements, then printing the last generated element and halting.

Exercise 5.21

If an infinite r.e. set has an increasing enumerating function, we can decide membership of an arbitrary string x in the set by enumerating the strings until one of two things happen: (i) x appears in the enumeration, in which case we can state that x belongs to the set; or (ii) the last string enumerated exceeds x in the ordering, in which case we can state that x does not belong to the set (because it can no longer appear in the enumeration, since that would cause a decrease). Since one of the two must happen eventually, we have correctly decided membership and the set is recursive.

If a set is recursive, then we can enumerate it in an increasing manner by the simple device of enumerating (internally) all possible strings in increasing order and deciding for each string as it is enumerated whether or not it belongs to the set; if it does, we print it.

Exercise 5.22

S is neither recursive nor r.e. That it is not recursive is an immediate consequence of Rice's theorem. Alternatively, we can reduce K to S as follows. Let i be the index of some not totally undefined function and define $\theta(x, y) = \text{Zero}(\phi_{\text{univ}}(x, x)) + \phi_i(y)$. Since θ is partial recursive, there exists some index k such that $\phi_k(x, y) = \theta(x, y)$ and, by the s-1-1 construction, there exists some index $l = s(k, x)$ such that $\phi_l(y) = \phi_k(x, y)$ for each fixed x. Now note that

$$\phi_l(y) = \phi_k(x, y) = \theta(x, y) = \text{Zero}(\phi_{\text{univ}}(x, x)) + \phi_i(y)$$

is totally undefined if $x \notin K$ and is the function ϕ_i otherwise. Thus we have $\{l, i\} \in S$ if and only if $x \in K$; that is, we can decide whether or not $x \in K$ by deciding whether or not $h(x) = \{s(k, x), i\} \in S$, where i and k are known constants. Since K is not recursive, neither is S.

Now, it would seem that the complement of S is r.e.: we can run, using the step function, a dovetailing computation on ϕ_i and ϕ_j to check, for each possible argument x, whether $\phi_i(x)$ and $\phi_j(x)$ are both defined and different, in which case we can immediately answer yes (i.e., print the pair $\{i, j\}$ as belonging to $\overline{S}$). The problem with that argument is that it ignores pairs $\{i, j\}$ that belong to $\overline{S}$ because one function is defined where the other is not (suppose, for instance, that ϕ_i is the totally undefined function), a difference that cannot be affirmed through enumeration. So, we should in fact conclude that neither S nor its complement is r.e., which we can prove by reductions from the set $\overline{K}$.

To prove that S is not r.e., we use exactly the same construction as in the previous reduction; now let j_0 be the index of the totally undefined function. Then we have $\{l, j_0\} \in S$ if and only if $x \notin K$ if and only if $x \in \overline{K}$. Hence membership in S and membership in $\overline{K}$ are equally hard to determine, so that S is not r.e. (If we could semi-decide S, we could also semi-decide $\overline{K}$ by computing $l = s(k, x)$ from x, then asking whether $\{l, j_0\}$ is in S; since we cannot semi-decide $\overline{K}$, neither can we semi-decide S.)

Proving that $\overline{S}$ is not r.e. was not asked. We can do it by the same reduction from $\overline{K}$. Remember that the resulting function, $\phi_{s(k,x)}$, is totally undefined if $x \in \overline{K}$ and is the not totally undefined function ϕ_i otherwise. Now the pair $\{i, s(j, x)\}$ is in $\overline{S}$ (i.e., $\phi_{s(j,x)}$ does not compute the same function as ϕ_i) if and only if $x \in \overline{K}$. Since $\overline{K}$ is not r.e., neither is $\overline{S}$.

Exercise 5.23

Both sets are non-recursive, although we cannot use Rice's theorem, since the property defining the sets is index-dependent (it is a property of programs but not of functions). So we prove that they are non-recursive by direct reduction from K. Define the function $\theta(x, y) = \text{Zero}(\phi_{\text{univ}}(x, x))$ (that will be for A; for B, we will use $\text{Succ}(\theta(x, y))$. This is clearly a partial recursive function and as such has an index, say j; we can then use the s-m-n theorem to obtain the new function $\phi_{s(j,x)}(y) = \theta(x, y)$. Now, if $x \in K$, then $\phi_{s(j,x)}(y)$ is the constant function 0 (or the constant function 1 if we take the successor), yet, if $x \notin K$, then it is the totally undefined function. Thus, if $x \notin K$, then $s(j, x)$ belongs neither to A nor to B, whereas, if $x \in K$, then $s(j, x)$ belongs to A (or to B if we used the successor). This establishes a reduction from K to A (resp., B) and thus proves that neither A nor B is recursive. On the other hand, both sets are r.e.: we can run a dovetailing process on all indices and all steps; whenever we find a z such that $\text{step}(\Pi_1(z), \Pi_1(z), \Pi_2(z)) = 1$ (1 for the set A, but 2 for the set B), we print $\Pi_1(z)$.

Now assume that there exists some recursive set C that separates A and B, i.e., such that $A \subseteq C$ and $B \subseteq \overline{C}$, and let ϕ_j (a total function) be its characteristic function. What is the value of $\phi_j(j)$? Since ϕ_j is a characteristic function, the value must be either 0 or 1. If the value is 0, this means that j is not in C; but note that we then also have $j \in A$ by definition of A; yet $A \subseteq C$, so we have a contradiction: j is in A which is inside C, yet j is not in C. Conversely, if the value is 1, then this means that j is in C; but we then also have $j \in B$ by definition of B; yet B is disjoint from C, so j cannot be both in C and in B, another contradiction. Hence there is no value that $\phi_j(j)$ can assume without causing a contradiction, so that ϕ_j cannot exist (at least not as a total recursive function) and thus C cannot be a recursive set.

Exercise 5.24

1. Note that the subscript i of ψ_i is *not* an index in a programming system; it is instead used simply to identify a certain function in a convenient manner. Indeed, it is to be viewed as the result of a pairing; we can write it $i = \langle n_i, i_0, i_1, \ldots, i_{n_i} \rangle$ or, when we just want to look at two pieces, $i = \langle n_i, w_i \rangle$. (We should view i as the pairing of a count n_i and of n_i items following the count, because of the specific use of the projection functions in the definition of f.) With this notation in mind, we see that $\psi_i(x)$ is defined whenever we have $x \leq n_i$ and undefined otherwise; when it is defined, say when $x = k$ for some k, $0 \leq k \leq n_i$, then its value is i_k. Now let g be some arbitrary function

from $\{0, 1, 2, \ldots, n\}$ to $\mathbb{N}$; consider the pairing

$$z = \langle n, g(0), g(1), g(2), \ldots, g(n)\rangle_{n+2}$$

It is clear that ψ_z is the function g, so that any arbitrary function from a finite initial subset of $\mathbb{N}$ to $\mathbb{N}$ has a corresponding ψ_i.
Since pairing functions are injective, $i \neq j$ implies $n_i \neq n_j$ or $w_i \neq w_j$. If $n_i \neq n_j$, then ψ_i and ψ_j are not defined on the same interval and thus are distinct; if $n_i = n_j$, then $w_i \neq w_j$ and thus there exists some index k, $0 \leq k \leq n_i$, such that $i_k \neq j_k$; but then $\psi_i(k) = i_k \neq j_k = \psi_j(k)$ and thus the two functions are distinct.
Hence each ψ_i corresponds to its own specific function from an initial subset of $\mathbb{N}$ to $\mathbb{N}$.

2. It is rather interesting that such a small change in the definition would generalize from initial subsets to any finite subsets. In order to encode a mapping ϕ from any finite subset of $\mathbb{N}$ to $\mathbb{N}$, we need to use either a sparse encoding, where only the defined values are encoded (but then they must be encoded as pairs, with the value on which the function is defined paired up with the value of the function—i.e., a pair of the form $\langle k, \phi(k)\rangle$), or a dense encoding, where the value of the function on every argument from 0 to the largest value on which the function is defined is represented. In the latter case, we must then adopt some convention for representing undefined function values. We have already seen one such convention with the step function: we represent undefined by 0 and add 1 to all defined values. This is the encoding adopted by the family $\{\pi_i\}$: note that they decrement the value returned by f through unpairing and are undefined whenever that value is 0.
More concretely, them we should think of the subscript i as a pairing of the form $i = \langle n_i, i_0, i_1, \ldots, i_{n_i}\rangle$ and of the values i_k as either 0 when the function is undefined on argument k or one more than the value of the function at k. In that light, n_i is simply the largest argument on which the function is defined.
Once this structure is understood, everything else follows easily as above. Once again, each π_i corresponds to its own specific function from a finite subset of $\mathbb{N}$ to $\mathbb{N}$.

Exercise 5.25

This is the proof of the Rice-Shapiro theorem. Since the theorem offers an exact characterization, our proof has two parts: we need to show that any set with the given characteristics is r.e. and, separately, that any r.e. sets has these characteristics.

Let us then assume that our set $S = \{x \mid \phi_x \in \mathscr{C}\}$ is such that there exists an r.e. set I such that $\phi_x \in \mathscr{C}$ holds if and only if there exists some $i \in I$ with $\pi_i \subseteq \phi_x$. Note that, given x and i, we can semi-decide $\pi \subseteq \phi_x$, because there are only finitely many arguments on which π_i is defined and we can run a dovetailing computation of ϕ_x on these arguments. If we get results that differ from π_i, we can answer no; if we get convergence and agreement on all values, we can answer yes; it is only in the case where ϕ_x diverges on one of the arguments on which π_i is defined (a case in which the answer should be no) that we are unable to decide. Thus we can decide all the "yes" instances—i.e., we can semi-decide. Now, the set I is itself r.e., so we can add one level of dovetailing: given some input x, we dovetail on all $i \in I$, all arguments on which π_i converges, and all steps, and accept x whenever we have found convergence and agreement for some π_i. Hence our set S is r.e.

For the converse, let us assume that the set $S = \{x \mid \phi_x \in \mathscr{C}\}$ is r.e. We want to define an r.e. set I with the required property; in other words, we want to enumerate a collection of π_i with the required property. We can use an s-1-1 construction to obtain a total function f with $\phi_{f(i)} = \pi_i$ for all $i \in \mathbb{N}$. Let ϕ be a domain function for S, i.e., such that $dom\phi = S$. We simply define $I = dom(\phi \cdot f)$. By definition, I is r.e.; by construction, we have $i \in I \Leftrightarrow \pi_i \in \mathscr{C}$. It remains that show that this property extends from the finite behavior π_i to the full function $\phi_j \in \mathscr{C}$.

We claim that we have

$$\phi_j \in \mathscr{C} \Leftrightarrow \exists i \in I,\ \pi_i \subseteq \phi_j$$

The ($\Rightarrow$) part is fairly simple: if ϕ_j is in $\mathscr{C}$, define the new function

$$\theta(x, y) = \begin{cases} \phi_j(y) & \forall z \leq y\ [\text{step}(x, x, z) = 0] \\ \text{undefined} & \text{otherwise} \end{cases}$$

Using an s-1-1 construction, we write $\phi_{g(x)}(y) = \theta(x, y)$. Note that, if x is in K, then $\phi_{g(x)}$ is a finite subfunction of ϕ_j, whereas, if x is not in K, then $\phi_{g(x)}$ equals ϕ_j. Now, if ϕ_j had no finite subfunction π_i in $\mathscr{C}$, then g would be a many-one reduction from $\overline{K}$ to S, contradicting the fact that S is r.e. (If x is in $\overline{K}$, then $\phi_{g(x)}$ equals ϕ_j, which is in S, but otherwise $\phi_{g(x)}$ has finite domain and thus cannot be in S.) Thus ϕ_j must have some finite subfunction π_i in $\mathscr{C}$, as desired.

The reverse direction is more subtle. We need to show that any partial recursive function that extends a function in $\mathscr{C}$ must also be in $\mathscr{C}$—perhaps the most fundamental property of r.e. sets! This property is true whether or not the function to be extended has finite domain. So let ϕ_j be some function in $\mathscr{C}$ and let ψ be some partial recursive function with $\phi_j \subseteq \psi$; we want to prove $\psi \in \mathscr{C}$. Define

$$\theta(x, y) = \begin{cases} \psi(y) & x \in K \\ \phi_j(y) & x \notin K \end{cases}$$

(We shall take up shortly how such a definition can be implemented within the partial recursive functions.) Using an s-1-1 construction, we get $\phi_{g(x)}(y) = \theta(x, y)$. Now, if ψ were not in $\mathscr{C}$, g would be a many-one reduction from $\overline{K}$ to S, contradicting the fact that S is r.e. (If x is in $\overline{K}$, then $\phi_{g(x)}$ equals ϕ_j, which is in S, but otherwise $\phi_{g(x)}$ equals ψ which is not in S.) Thus ψ must be a function in $\mathscr{C}$, as desired.

To finish our proof, we need to show that the definition of θ just given is partial recursive. Basically, the idea is to dovetail a computation of $\phi_j(y)$ and an enumeration of K. If $\phi_j(y)$ converges before x appears in the enumeration of K, we take advantage of the fact that ψ agrees with ϕ_j everywhere the latter is defined and return $\phi_j(y)$, which is also $\psi(y)$—although we have not determined which case applies, it does not matter. If x turns up in the enumeration of K before $\phi_j(y)$ converges, then we stop computing $\phi_j(y)$ and start computing instead $\psi(y)$ and let that computation define $\theta(x, y)$. Specifying the step functions and the dovetailing formally is a big mess, so we leave it at this intuitive level.

Exercise 5.26

1. Yes, such an index n exists. Define the function

$$\theta(x, y) = \begin{cases} 0 & y = x^2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

   Using an s-1-1 construction, there exists a total function f such that $\phi_{f(x)}(y) = \theta(x, y)$. By the recursion theorem, f has a fixed point n such that $\phi_n = \phi_{f(n)}$. But then we have $\phi_n(y) = \phi_{f(n)}(y) = \theta(n, y)$ and thus $\phi_n(y)$ is defined only $y = n^2$. Thus the index n is the desired answer.

2. Yes, such an index n exists. Define the function

$$\theta(x, y) = \begin{cases} 0 & x \neq y \\ \text{undefined} & x = y \end{cases}$$

Using an s-1-1 construction, there exists a total function f such that $\phi_{f(x)}(y) = \theta(x, y)$. By the recursion theorem, f has a fixed point n such that $\phi_n = \phi_{f(n)}$. But then we have $\phi_n(y) = \phi_{f(n)}(y) = \theta(n, y)$ and thus $\phi_n(y)$ is defined everywhere except for $y = n$—in other words, its domain is $\mathbb{N} - \{n\}$. Thus the index n is the desired answer.

3. Yes, such an index n exists. Define the function

$$\theta(x, y) = \begin{cases} 0 & y = x \\ \phi_{\text{univ}}(y, y) & \text{otherwise} \end{cases}$$

Using an s-1-1 construction, there exists a total function f such that $\phi_{f(x)}(y) = \theta(x, y)$. By the recursion theorem, f has a fixed point n such that $\phi_n = \phi_{f(n)}$. But then we have $\phi_n(y) = \phi_{f(n)}(y) = \theta(n, y)$ and thus $\phi_n(y)$ is defined for $y = n$ as well as for any value of y in K and undefined everywhere else. other words, its domain is $K \cup \{n\}$. Thus the index n is the desired answer.

Exercise 5.27

We need to develop a reduction from the known productive set $\overline{K}$ to the set $S(c)$ that allows us to derive the productive function for $S(c)$. Thus what we want is a reduction such that, if x is in $\overline{K}$, i.e., if $\phi_x(x)\uparrow$, then $x \in S(c)$, i.e., $\phi_x(c)\uparrow$. Define the function $\theta(x, y) = \phi_{\text{univ}}(x, x) - \phi_{\text{univ}}(x, x)$. By standard s-m-n arguments, there exists some total function f such that $\phi_{f(x)}(y) = \theta(x, y)$. But note that $\phi_{f(x)}$ is either the constant function ϵ (when $x \notin \overline{K}$), in which case $f(x) \notin S(c)$, or the totally undefined function (when $x \in \overline{K}$), in which case $f(x) \in S(c)$. Thus we have a reduction from $\overline{K}$ to $S(c)$ such that $x \in \overline{K} \Leftrightarrow f(x) \in S(c)$. (Incidentally, this reduction shows that $S(c)$ is non-r.e., not surprisingly, since every productive set is by definition non-r.e.) Now note that, if we have $dom\phi_i \subseteq S(c)$, then we have $f^{-1}(dom\phi_i) \subseteq f^{-1}(S(c)) = \overline{K}$. (We use $f^{-1}(A)$ to denote the inverse image of A under f, i.e., the set of all x such that $f(x) \in A$; we do not use f^{-1} to denote any kind of function!) Moreover, $f^{-1}(dom\phi_i)$ is r.e., so that there exists some ϕ_z such that $dom\phi_z = f^{-1}(dom\phi_i)$; to see that, just note that $\phi_z(y) = \phi_i(f(y))$ has the desired properties. But then $dom\phi_z \subseteq \overline{K}$ and so, by our previous remarks, $z \in \overline{K} - dom\phi_z$; it follows that $f(z) \in S(c) - dom\phi_i$.

So all we need is to compute the z, i.e., to find a function g such that $z = g(i)$; then the productive function for $S(c)$ is just $f \cdot g$. We saw that $\phi_z(y) = \phi_i(f(y))$ has the desired properties. Thus, if f is the (total) function ϕ_j, then we have $z = c(i, j)$ (using the composition function c) and the productive function for $S(c)$ is just $p_{S(c)}(i) = \phi_j(c(i, j))$, where j is a known constant.

Exercise 5.28

We are going to use the productive function f to enumerate an infinite subset of the productive set A. We proceed inductively. We can start with the totally undefined function; let i be the index of a program for that function. Since the empty set is a subset of A, we have $dom\phi_i\emptyset \subseteq A$ and thus also $f(i) \in A - dom\phi_i = A$. We let that be the first element of our infinite set, $x_0 = i$. Now assume that we have generated elements $x_0, x_1, \ldots, x_n$ of our new set, in that order, and let us generate element x_{n+1}. There is a partial recursive function with domain exactly equal to $\{x_0, x_1, \ldots, x_n\}$, call it ϕ_{j_n}—how to generate this function is a problem we shall tackled shortly. Then we have $dom\phi_{j_n} \subseteq A$ and thus $f(j_n) \in A - dom\phi_{j_n}$; in other words, $f(j_n)$ is an element of A not in the set $\{x_0, x_1, \ldots, x_n\}$, and so is exactly the new element we wanted, $x_{n+1} = f(j_n)$. Since we have a recursive procedure to enumerate an infinite subset of A, the subset is r.e., as desired.

That the function ϕ_{j_n} can be constructed may at first appear difficult, but it, too, can be constructed recursively. We have the first such function, the totally undefined function. Assume that we have already produced the function $\phi_{j_{n-1}}$ with domain $\{x_0, x_1, \ldots, x_{n-1}\}$; we can define ϕ_{j_n} as follows:

$$phi_{j_n}(y) = \begin{cases} 1 & y = f(j_{n-1}) \\ \phi_{j_{n-1}}(y) & \text{otherwise} \end{cases}$$

and a standard s-1-1 construction produces the index j_n from the index j_{n-1}.

Exercise 5.29

1. To show that a set and its cylindrification belong to the same m-degree, we need to demonstrate many-one reductions from each to the other. Reducing s to $S \times \mathbb{N}$ is trivial: we can simply map each value x to the pair $(x, 0)$. Clearly, x belongs to S if and only if $(x, 0)$ belongs to $S \times \mathbb{N}$. The converse is equally easy: given a pair (x, y), we map it to the value x; again, it is clear that (x, y) belongs to $S \times \mathbb{N}$ if and only if x belongs to S.
2. We know that simple sets are not creative. This part of the problem then asks us to show that this property is preserved through cylindrification. (But simplicity may not be: we are not asked to show that $S \times \mathbb{N}$ is simple, only that it is not creative—in fact, it is not simple.) Let S be a simple set and $S \times \mathbb{N}$ its cylindrification. Since S is r.e., so is $S \times \mathbb{N}$ (any enumerator for S can be turned into an enumerator

for $S \times \mathbb{N}$ through simple dovetailing); thus we need to show that the complement of $S \times \mathbb{N}$, which is just $\overline{S} \times \mathbb{N}$, is not productive.

Assume then that $\overline{S} \times \mathbb{N}$ is productive with productive function f. We want to show that we can use f to construct a productive function for $\overline{S}$, which would contradict the simplicity of S and prove our result. Let ϕ be some partial recursive function with $dom\phi \subseteq \overline{S}$; then the function $\theta(\langle x, y\rangle) = \phi(x)$ clearly obeys $dom\theta \subseteq S \times \mathbb{N}$; moreover, we have $dom\theta = dom\phi \times \mathbb{N}$. If we write $\phi_j(\langle x, y\rangle) = \theta(\langle x, y\rangle)$, then we have $f(i) \in (\overline{S} \times \mathbb{N}) - dom\theta$, that is, $f(i) \in (\overline{S} \times \mathbb{N}) - (dom\phi \times \mathbb{N})$, that is $f(i) \in (\overline{S} - dom\phi) \times \mathbb{N}$. But $f(i)$ is some pair $\langle x, y\rangle$, so that $\Pi_1(x) \in \overline{S} - dom\phi$. In other words, $\Pi_1 \cdot f$ is a productive function for $\overline{S}$, the desired contradiction.

Exercise 5.30

This is a rather open-ended exercise. To circumscribe it, let us consider the various results of Section 5.7.

The first result (Proposition 5.2) was that there is a unique m-degree that contains exactly the nontrivial recursive sets. As mentioned in the problem statement, such is no longer the case with 1-degrees: for instance, in order for two finite sets to reduce to each other through one-reductions, they must be of the same size. Let us then prove what is given in the problem statement: namely that the recursive sets are partitioned into the following equivalent classes: finite sets (one class for each size), infinite sets with finite complements (again, one class for each size of complement), and infinite sets with infinite complements (a single class). In order to reduce finite set A to finite set B with an injective map, we must have $|A| \leq |B|$; thus in order for the two sets to be in the same 1-degree, we must have $|A| = |B|$. Now, if two finite sets have the same size, we claim that we can always reduce one to the other: basically, we can order the elements of the universe and define a map that is essentially an identity, except on the elements that belong to one of the two sets; we map the elements of A to those of B following the ordering and map the elements of $B - A$ to elements of $A - B$, again according to the ordering. Thus all finite sets of the same size form a 1-degree. Essentially the same argument shows that infinite sets with finite complements are partitioned the same way, according to the size of their complement (the mapping is one-to-one on the universe and thus in particular on the complements, with exactly the consequences described above). Finally, we claim that infinite recursive sets with infinite complements form a single equivalence class: we can enumerate each set and each complement and define the mapping through the enumeration—map the ith element of the first set to the ith element of the second and the j element of the complement of the first set to the jth element of the complement of the second. In terms of lattice of 1-degrees, we see that we can one-reduce a small finite set to a larger finite set, but not vice-versa; similarly, we can reduce an infinite set with a small finite complement to an infinite set with a larger finite complement, but not vice-versa; finally, we can reduce either type of set to an infinite set with infinite complement. Thus the lattice has a single element at the top, the 1-degree of infinite recursive set with infinite complement and has two infinite "columns" underneath, one formed by 1-degrees of finite sets, with size 0 at the bottom and increasing by one at each level, and one formed by 1-degrees of infinite sets with finite complements, with complement size 0 at the bottom and complement size increasing by one at each level. Comparing with Figure 5.5, we see that both have $\{\emptyset\}$ and $\{\mathbb{N}\}$ at the bottom, but that many-one reductions merge all non-zero sizes (finite or infinite).

Our next result is Proposition 5.3, which states that an m-degree that contains an r.e. set contains only r.e. sets. That result is completely unaffected by the nature of the reduction, since it uses only composition of functions.

Our definition of many-one completeness is changed into the obvious definition of one-completeness. Theorem 5.8 remains true with the change: K is one-complete for the class of r.e. sets: this is due to an earlier observation that the s-m-n function is in fact an injective function (our Theorem 5.1 constructs it mostly through pairing, which is injective). Proposition 5.4, on the other hand, is clearly false for on-completeness, in view of the lattice we just discussed; it needs to be modified to "any infinite recursive set with infinite complement is one-complete for the class of recursive sets." *Ditto* with the comment just below: only infinite recursive sets with infinite complements one-reduce to their complement.

The discussion of productive, creative, and simple sets is not changed, since their definitions do not involve a specific type of reduction. Thus Theorem 5.10 remains unchanged. Somewhat surprisingly, so does Theorem 5.9: we can in fact state that an r.e. set if many-one complete for the r.e. sets if and only if it is one-complete for them and if and only if it is creative. The "only if" part of the proof is clearly unaffected: creativity does not depend on the type of reduction used. The "if" part is the interesting one, since we need to show that the induced reduction $f \cdot e$ is injective. Once again, though, this follows rather directly from the fact that the s-m-n function is injective. The least-upper-bound argument of Theorem 5.11 clearly remains unaffected, since the mappings used are injective.

Finally, we want to consider cylindrifications. We proved in Exercise 5.29 that a set and its cylindrification belong to the same m-degree, but only one of the two mappings used in that proof was injective. Indeed, it is pretty clear that a finite set and its cylindrification cannot belong to the same 1-degree, since we cannot one-reduce the infinitely large cylindrification to the finite set. The cylindrification of any nontrivial recursive set is an infinite recursive set with infinite complement and thus sits at the top of the lattice. Thus exceptions can only arise at the top or at the very bottom of the lattice. The cylindrification of the empty set is still empty and the cylindrification of the universe remains the universe; in those cases, the set and its cylindrification do belong to the same 1-degree at the very bottom of the lattice. And the cylindrification of an infinite recursive set with infinite complement is another infinite recursive set with infinite complement, so that these sets and their cylindrifications do belong to the same 1-degree, at the very top of the lattice.

Solution Manual
Chapter 5, Additional Problems

Additional Problem 1

1. Denote the least common multiple of x and y by lcm(x, y); since the lcm of x and y cannot exceed xy, we simply write

$$\text{lcm}(x, y) = \min z \leq \text{mult}(x, y)\ [x|z \text{ and } y|z]$$

2. Denote the greatest common divisor of x and y by gcd(x, y); we simply write

$$\text{gcd}(x, y) = \max z \leq x\ [z|x \text{ and } z|y]$$

3. Let the number of divisors of x be $d(x)$ and the number of prime divisors of x be $pd(x)$. In both this problem and the next, we need a primitive recursive construct analogous to a bounded search that returns the number of distinct $z \leq x$ that meet a certain predicate. We define such a construct below, denoting it count$(z \leq x)[P(z)]$. Then we have

$$d(x) = \text{count}(z \leq x)[z|x]$$

and

$$pd(x) = \text{count}(z \leq x)[(z|x) \text{ and is\_prime}(z)]$$

To define the count schema, we use primitive recursion as follows (assuming that P is a primitive recursive predicate):

$$\begin{cases} \text{count}(z \leq 0)[P(x)] = P(x) \\ \text{count}(z \leq x+1)[P(x)] = \text{add}(P(x+1), \text{count}(z \leq x)[P(x)] \end{cases}$$

4. Euler's function is generally written $\phi(x)$. Using the count construct, we write

$$\phi(x) = \text{count}(z \leq x)[gcd(z, x) = 1]$$

5. Denote the leading nonzero bit in the unsigned binary representation of x by $\text{msb}(x)$. We write

$$\begin{cases} \text{msb}(1) = 0 \\ \text{msb}(x+1) = \begin{cases} \text{Succ}(\text{msb}(x)) & \text{is\_power\_of\_2}(x+1) \\ \text{msb}(x) & \text{otherwise} \end{cases} \end{cases}$$

and define

$$\begin{cases} \text{is\_power\_of\_2}(1) = 1 \\ \text{is\_power\_of\_2}(x+1) = \left[x + 1 = \text{mult}(2, \max z \le x\ [\text{is\_power\_of\_2}(z)])\right] \end{cases}$$

Additional Problem 2

Classify each of the following sets and its complement as recursive, nonrecursive r.e., or non-r.e.

1. We can certainly produce the decimals of π, one after the other; since the decimals are enumerable, we can definitely semi-decide the set at hand: just proceed with the enumeration of the decimals of π, watching runs of consecutive 7s as we do so. As soon as a run of x 7s is found, stop and answer "yes." But we can do better, although not constructively: this set is in fact decidable. Either π has a longest consecutive run of 7s or it does not. (An example of an irrational number that does not have a longest run of consecutive 7s is $0.707707770777707777770\ldots$.) If it does not, our set is just $\mathbb{N}$, which is certainly decidable! If it does, then let n_0 be the length of that longest consecutive run; but then our set is just $\{0, 1, 2, \ldots, n_0\}$, which is certainly decidable (just ask "is x larger than n_0?). We do not know which of the two cases is the correct one, but one of them is and thus there exists an algorithm that correctly decides membership in our set. Which of the two algorithms is the correct one will require further insights in number theory, but we can state that the problem is decidable.
2. The difference between this case and the previous one is that we cannot assume that every value of x below a certain limit (or every value at all) belongs to the set. If π does not have a longest run of consecutive 7s and thus has an infinite number of runs of different lengths, it need not have runs of every possible length; thus deciding membership in our set would not be trivial. On the other hand, if π does have a longest run of consecutive 7s, say of length n_0, then our set is definitely decidable, since we could build a table showing, for each

$i \leq n_0$, whether π has a run of consecutive 7s of length i. Again, we do not have enough results in number theory to be able to build such a table today, but such a table exists, and hence a decision algorithm exists.

So we can say this: if π has a longest run of consecutive 7s, then the problem is decidable; otherwise, the problem may or may not be decidable. Thus, at present, all we can say is that our set is r.e.; we lack information to say anything about its recursiveness.

3. In words, this is the set of all functions that can always produce the double of whatever output they produce; in particular, any such function has infinite domain or is totally undefined or only returns 0. Since this property is strictly an I/O property, Rice's theorem tells us that neither the set nor its complement is recursive. In fact, neither one is r.e., as we proceed to prove with reductions from $\overline{K}$.

   To prove that S is not r.e., we define the function

$$\theta(x, y) = \text{Succ}\big(\text{Zero}(\phi_{\text{univ}}(x, x))\big)$$

   By our usual arguments, there exists $s(x)$ such that $\phi_{s(x)}(y) = \theta(x, y)$. Now, if $x \in \overline{K}$, then $\phi_{s(x)}(y)$ is totally undefined and thus $s(x) \in S$—because the predicate holds vacuously. On the other hand, if $x \in K$, then $\phi_{s(x)}(y)$ is the constant function Succ(Zero(y)), always returning 1 for any argument, and thus $s(x) \notin S$, as desired (because there is no way to double the 1).

   Proving that $\overline{S}$ is not r.e. is tricky; we define the function θ as follows

$$\theta(x, y) = \begin{cases} 1 & step(x, x, y) = 0 \\ \Pi_1(y) & \text{otherwise} \end{cases}$$

   By our usual arguments, there exists $s(x)$ such that $\phi_{s(x)}(y) = \theta(x, y)$. Now, if $x \in \overline{K}$, then $\phi_{s(x)}(y) = 1$, thus $s(x) \in \overline{S}$, since the 1 cannot be doubled. On the other hand, if $x \in K$, then $\phi_{s(x)}(y)$ is 1 for $y < y_0$ (y_0 is some unknown constant) and $\Pi_1(y)$ for all $y \geq y_0$, so that the range of $\phi_{s(x)}$ is all of $\mathbb{N}$ (because the unpairing ensures that, no matter what the constant y_0, all natural integers will be encountered again: for any integer i, there exists an integer $y > y_0$, for instance $y = \langle i, y_0 \rangle$, such that $\Pi_1(y) = i$), so that $s(x) \in S$, as desired.

   Another possibility is to let

$$\theta(x, y) = \begin{cases} 1 & y = 0 \\ \text{Succ}(y) + \text{Zero}(\phi_{\text{univ}}(x, x)) & \text{otherwise} \end{cases}$$

then use the s-m-n theorem on θ to obtain $\phi_{s(x)}(y) = \theta(x, y)$. Note that, if $x \in \overline{K}$, then $\phi_{s(x)}(y)$ is only defined for $y = 0$ and has value 1, which cannot be doubled, so that $s(x) \in \overline{S}$, as desired. On the other hand, if $x \in K$, then $\phi_{s(x)}(y) = y + 1$ and so can double any value it generates, so that $s(x) \in S$.

4. This is a non-r.e. set. Informally, the fact that a machine accepts an infinite language cannot be verified, since we can only verify a finite number of acceptances, so that we cannot even semi-decide the set.
   We reduce the known non-r.e. set $\{x \mid M_x$ always halts$\}$ to our set. What we want to do is to take M_x and transform it into a new Turing machine M'_x such that M'_x accepts an infinite language if and only if M_x always halts. The problem is those machines M_x that do not always halt, yet accept an infinite language; otherwise, the rest is trivial. An easy solution is the following: M'_x, on input z, will run M_x on all strings less than z, one after the other; after running all of them and z itself, the machine erases the tape and prints a 1. Now, if M_x always halts, M'_x will accept Σ^*, an infinite language. On the other hand, if M_x does not always halt, then there is some smallest string y on which M_x does not halt and we can see that M'_x will halt on all strings less than y and fail to halt on y and on all strings larger than y—in other words, M'_x will only halt on a finite number of strings. Thus our reduction works correctly: M_x always halts if and only if M'_x accepts an infinite language.
5. Intuition about this set is harder to come. Given an enumerating machine M, how do we know whether the language that it enumerates is equal to its reverse? We can try running the enumerator (through dovetailing) and keeping track of strings whose reverse has not yet been generated, but we can never be sure that we have seen all strings in the language generated by M, and so can never decide to accept M. This line of thought seems to indicate that the set is non-r.e.
   We reduce the set $\{x \mid M_x$ always halts$\}$ to our set. Given M_x, we produce a new machine M'_x which does the following: when given input z, it runs M_x on all strings from ϵ up to y, then erases the tape and outputs a 1. As in our previous reduction, M'_x will accept Σ^* (a language equal to its reverse) if M_x halts under all inputs. However, if M_x does not always halt, let y be the smallest string under which it does not halt; then M'_x accepts all strings smaller than y and enters an infinite loop if presented with any string equal to or larger than y. Now we want this finite language accepted by M'_x in the second case not to be equal to its reverse in order to have a valid reduction; when could the language of all strings in lexicographic order from ϵ

up to (but not including) some y be equal to its reverse? Because of the lexicographic order, only when the string y is one character longer than its predecessor in the order—that's because the first string of length n generated in lexicographic order has as reverse the last string of length n generated in lexicographic order. So we need only modify our machine M'_x a little bit: when presented with a string y that is the first string in lexicographic order, it should accept it without running M_x on it. Now M'_x actually accepts an infinite language no matter what (at least one string for each length n, the first of that length in lexicographic order), but only accepts a language equal to its reverse if it accepts all strings. Thus our reduction is complete.

Additional Problem 3

1. The statement is false. One intuitive way to see why is to look at the equivalent definition of XOR: it involves intersection and union (under which r.e. sets are closed) and subtraction (or complement), under which r.e. sets are not closed. A simple counterexample is given by $S = \mathbb{N}$ and $T = K$; then the exclusive-or of the two is $\overline{K}$, which is not r.e.
2. The statement is true. If S or T is empty, then so is U, which is thus r.e., as desired. Assume then that S and T are non-empty; then they have total enumeration functions f and g. We claim that

$$h(z) = f(\Pi_1(z)) + g(\Pi_2(z))$$

is an enumeration function for Us, i.e., that $ranh = U$. Assume that $x \in ranh$; then there exists z with $x = h(z) = f(\Pi_1(z)) + g(\Pi_2(z)) = a + b$, where $a \in S$ and $b \in T$, since f enumerates S and g enumerates T; thus $ranh \subseteq U$. Now assume that $x \in U$; then we can write $x = a + b$, where $a \in S$ and $b \in T$; but then there exist y_1 and y_2 such that $f(y_1) = a$ and $g(y_2) = b$, since f enumerates S and g enumerates T; writing $z =< y_1, y_2 >$, we now observe that $h(z) = f(\Pi_1(z)) + g(\Pi_2(z)) = f(y_1) + g(y_2) = a + b = x$, so that $x \in ranh$; thus $U \subseteq ranh$.
3. The statement is false. The sets defined in Exercise 5.19, namely the collection of sets $\{\overline{K_t} \mid t \in \mathbb{N}\}$, give us the desired counterexample. Exercise 5.19 shows that the intersection of all such sets is $\overline{K}$, which is not r.e. and thus also not recursive. All of the $\overline{K_t}$ sets are recursive, however. Thus it remains only to show that we can design a recursive set S such that, for each of the $\overline{K_t}$, S contains at least one program that serves as characteristic function for that $\overline{K_t}$. The following program serves as characteristic function for $\overline{K_t}$:

```
if step(x,x,t) > 0
   then print 0
   else print 1
```

(under the usual assumption about step functions). For each t, we can easily compute the index of this program—in fact, we can write a total recursive function that, given the value of t, outputs the index of this program; moreover, the index grows as t grows. We let S be the set of these indices for all values of t. Given some arbitrary index i, we can decide if it is the index of one of these programs—just generate indices for increasing values of t until either one index equals i or one index exceeds i. Hence S is recursive. Now we have identified a recursive set containing only characteristic functions and such that the intersection of the recursive sets these functions define is non-recursive—our desired counterexample.

Additional Problem 4

We use the non-repeating enumerator for S, say f_S. We know that the set $K = \{x \mid M_x \text{ halts on } x\}$ is r.e., but non-recursive. Define the new set $S' = f(K)$—i.e., let the set S' be composed of those elements of S that are enumerated in ith position, where i is such that $M_i(i)$ is defined. By construction, S' is a subset of S. It is non-recursive, because K is non-recursive: otherwise, we could, given some element x, decide whether it belongs to K by mapping it through f to an element of S' and deciding membership in S'. Finally, S' is r.e., because it is produced by an enumerator in the first place (enumerate K and feed that through f).

Additional Problem 5

Given some infinite recursive (and hence also r.e.) set S, let V be a non-recursive r.e. subset of S (which must exist by the result of Additional Problem 5.4). Now consider $S - V$: we claim that it is non-r.e. If it were r.e., then we could decide V (a contradiction, since V is non-recursive) as follows: given an arbitrary element x, first ask whether it belongs to S; if it does not, answer "no"; if it does, then start enumerating V and $S - V$; since x must be in one of the two sets, it will eventually appear on one of the two lists and we can answer appropriately.

Additional Problem 6

Let c_A be the characteristic equation of A. Then $f^{-1}(A)$ has characteristic equation $c_A \cdot f$, since we need only map the element through f and ask if the result is in A. Since F is total, so is $c_A \cdot F$, so we are done.

We can enumerate $f(A)$ by the simple expedient of running f on A; that is, $f \cdot c_A$ has $f(A)$ has its range; since $f \cdot c_A$ is a recursive function, its range is r.e.

B is known to be r.e. Let ϕ_B be a function with $ran\phi_B = B$; then note that the function $f \cdot \phi_B$ has range $f(B)$, so that $f(B)$ is r.e. Finally, let ψ_B be a function with $dom\psi_B = B$; then note that the function $\psi_B \cdot f$ has domain $f^{-1}(B)$, which is thus r.e.

Additional Problem 7

Let us give a reduction-based proof rather than a direct proof. We prove that productivity is preserved under many-one reductions; then we reuse the construction from our proof of Rice's theorem to show that $\overline{K}$, a known productive set, reduces to the complement of any r.e. set obeying the hypothesis of Rice's theorem, thereby proving our result.

Let A be a productive set with productive function f and let h be a many-one reduction from A to B. Then it is easily seen that $h \cdot f$ is a productive function for B, since we have $x \in A \Leftrightarrow h(x) \in B$.

Now consider the construction of Rice's theorem. If the family of functions $\mathscr{C}$ contains the totally undefined function ϕ_u, then we claim that $\overline{K}$ reduces to our set $P_{\mathscr{C}} = \{x \mid \phi_x \in \mathscr{C}\}$, which is then productive and thus not r.e. It follows that our r.e. family of functions cannot contain the totally undefined function, so that its complement does contain the totally undefined function, so that, by the same reasoning, $P_{\overline{\mathscr{C}}}$ is productive, so that $P_{\mathscr{C}}$ is creative, as desired.

To verify the key claim, let ψ be some function not in $\mathscr{C}$ and let us define the function

$$\theta(x, y) = \psi(y) + \text{Zero}(\phi_{\text{univ}}(x, x))$$

and use an s-1-1 construction to obtain a function f such that $\phi_{f(x)}(y) = \theta(x, y)$. Now, if x belongs to $\overline{K}$, then $\theta(x, y)$ is totally undefined, so that $f(x)$ belongs to $P_{\mathscr{C}}$; conversely, if x belongs to K, then $\theta(x, y)$ is the function $\psi(y)$, so that $f(x)$ does not belong to $P_{\mathscr{C}}$. Hence we have reduced $\overline{K}$ to any nontrivial set of indices for a family of functions that includes the totally undefined function, showing that such a set of indices is productive.