CS5311 Homework #2 Solution (Spring 2002)

Due: Tuesday, February 19, 2002 **before class**

100 points

1. Do the following problems:

   (a) **[5 points]** Show that function $f(x) = \lfloor \log_2(x+1) \rfloor$ is primitive recursive.
   Answer: Let $t = \lfloor \log_2(x+1) \rfloor$. Then, we have

   $$t = \lfloor \log_2(x+1) \rfloor \leq \log_2(x+1) < t+1$$

   Therefore, $t+1$ is the smallest number that is larger than $\log_2(x+1)$. This is equivalent to say that $t+1$ is the smallest number such that $\lfloor \log_2(x+1) \rfloor < t+1$ holds. Consequently, $t+1$ is the smallest number such that $x+1 \leq 2^{t+1}$ holds. Since $\log_2(x)$ is a strictly increasing function, we have $t \leq \log_2(x+1) < t+1 < x+1$. This converts to the following:

   $$f(x) = \lfloor \log_2(x+1) \rfloor = \min_{t \leq x+1}[2^{t+1} \geq x+1]$$

   Since bounded minimalization, exponential, addition, and $\geq$ are all primitive recursive, so is f. ■

   (b) **[5 points]** Define predicate $R(x,y)$ as follows:

   $$R(x,y) = \begin{cases} 1 & \text{if } x \text{ and } y \text{ are relatively prime} \\ 0 & \text{otherwise} \end{cases}$$

   Two numbers x and y are said to be *relatively prime* if they have no common factor except 1. Show that $R(x,y)$ is primitive recursive *without* using quotient, remainder, and the GCD() function.
   Answer: If x and y are *not* relatively prime, they have a common factor that is greater than 1. That is, there exists a $t > 1$ such that $t|x$ and $t|y$ both hold. The value of t must be smaller than or equal to x or y. This translates to

   $$x \text{ and } y \text{ are } \textit{not} \text{ relativly prime} \Leftrightarrow \exists_{t \leq x}((t > 1) \ \& \ (t|x) \ \& \ (t|y))$$

   Therefore, we have

   $$x \text{ and } y \text{ are relatively prime} \Leftrightarrow \neg\,(\exists_{t \leq x}((t > 1) \ \& \ (t|x) \ \& \ (t|y)))$$

   Since $\neg$, $>$, $|$, &, bounded $\exists$ and two-way selection are all primitive recursive, so is $R(x,y)$. ■

   (c) **[5 points]** Show that the least common multiple function $\text{LCM}(x,y)$ is primitive recursive. The LCM of x and y is the smallest common multiple of x and y.
   Answer: For simplicity, let me ignore the special cases (*e.g.*, one of x or y being zero). The LCM of x and y is the *smallest* number that is a multiple of both x and y. A common multiple of x and y is a t satisfying $x|t$ and $y|t$. Since $x \times y$ is also a common multiple of x and y, the LCM must be smaller than or equal to $x \times y$. Consequently, the LCM is simply

   $$\text{LCM}(x,y) = \min_{t \leq x \times y}[(x|t) \ \wedge \ (y|t)]$$

   Since $|$, $\wedge$, $\times$ and bounded minimalization are primitive recursive, $\text{LCM}(x,y)$ is also primitive recursive. You can use two-way or multi-way selection to handle the cases of x or y being 0. ■

(d) **[5 points]** Show that the greatest common factor function GCD(x, y) is primitive recursive using the fact that LCM(x, y) is primitive recursive. The GCD of x and y is the largest common factor of x and y.

Answer: In your high school algebra textbook, you perhaps notice the relation of $x \times y =$ LCM $\times$ GCD. Therefore, we have

$$\mathrm{GCD}(x, y) = \lfloor (x \times y)/\mathrm{LCM}(x, y) \rfloor$$

and GCD(x, y) is primitive recursive. You can use two-way or multi-way selection to handle special cases. ■

2. **[10 points]** Let $\sigma(x)$ be the sum of the divisors of x if $x \neq 0$, and $\sigma(x) = 0$ if $x = 0$. For example, $\sigma(6) = 1 + 2 + 3 + 6 = 12$ because 1, 2, 3 and 6 are divisors of 6. Show that $\sigma(x)$ is primitive recursive.

Answer: Since $\sigma(x)$ is the sum of all factors of $x > 0$, we could rewrite it into the following expression:

$$\sigma(x) = \sum_{i=1}^{x} (i \times (i|x))$$

Thus, if i is a factor of x, the value of i is added to the sum. Since the largest divisor of x is x itself, the upper bound of the summation is x. Finally, a two-way selection is needed to handle the case of $x = 0$:

$$\sigma(x) = \begin{cases} 0 & \text{if } x = 0 \\ \sum_{i=1}^{x} i \times (i|x) & \text{otherwise} \end{cases}$$

Since $\times$, $\sum_{i=1}^{x}$, $x = 0$, $i|x$, and two-way selection are all primitive recursive, so is $\sigma(x)$. ■

3. Do the following problems:

(a) **[15 points]** The *history function* of $f(n)$, denoted as $f^*(n)$, is defined as follows:

$$\begin{aligned} f^*(0) &= 1 \\ f^*(n+1) &= [f(0), f(1), \ldots, f(n)] \\ &= 2^{f(0)} 3^{f(1)} \cdots p_n^{f(n)} \\ &= \prod_{i=0}^{n} p_i^{f(i)} \end{aligned}$$

Here, for the sake of convenience, we use $p_0 = 2$, $p_1 = 3$ and so on. Let $g(n)$ be a primitive recursive function such that $f(n) = g(f^*(n))$ holds. Show that $f^*(n)$ is primitive recursive, and then show that $f(n)$ is also primitive recursive.

Answer: By the definition of $f^*(n+1)$, we have

$$\begin{aligned} f^*(n+1) &= [f(0), f(1), \ldots, f(n)] \\ &= 2^{f(0)} 3^{f(1)} \cdots p_n^{f(n)} \\ &= \left(2^{f(0)} 3^{f(1)} \cdots p_{n-1}^{f(n-1)}\right) \cdot p_n^{f(n)} \\ &= f^*(n) \cdot p_n^{f(n)} \end{aligned}$$

Since $f(n) = g(f^*(n))$, the above can be rewritten as

$$f^*(n+1) = f^*(n) \cdot p_n^{g(f^*(n))}$$

Then, the given definition of $f^*(n)$ becomes

$$\begin{aligned} f^*(0) &= 1 \\ f^*(n+1) &= f^*(n) \cdot p_n^{g(f^*(n))} \end{aligned}$$

If we can show the second equation can be rewritten to a form of $F(n, f^*(n))$, $f^*()$ is primitive recursive. This is not difficult. Consider function $F(u, v)$ defined as follows:

$$F(u, v) = v \times p_u^{g(v)}$$

Since $\times$, p_u, x^y and $g()$ are all primitive recursive, $F(u, v)$ is also primitive recursive. Since f^* has a form of

$$\begin{aligned} f^*(0) &= 1 \\ f^*(n+1) &= F(n, f^*(n)) \end{aligned}$$

$f^*(n)$ is primitive recursive. Finally, since $f(n) = g(f^*(n))$ and since both $f^*()$ and $g()$ are primitive recursive, f is also primitive recursive. ∎

(b) **[5 points]** The Fibonacci sequence is defined as follows:

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ f(n-1) + f(n-2) & \text{if } n \geq 2 \end{cases}$$

Show that function $f(n)$ is primitive recursive. (**<u>Hint:</u>** use the history function of $f(n)$)
<u>Answer:</u> The history function of f is

$$\begin{aligned} f^*(0) &= 1 \\ f^*(n+1) &= [f(0), f(1), \ldots, f(n)] \end{aligned}$$

If we can find a primitive recursive function g such that $f(n) = g(f^*(n))$ holds, f and f^* are both primitive recursive. To compute $f(n)$, we need $f(n-1)$ and $f(n-2)$, the last two terms of $f^*(n)$. Since the last term is the $\text{Len}(f^*(n))$-th and the term next to the last is the $(\text{Len}(f^*(n)) - 1)$-th, we have

$$f(n) = f(n-1) + f(n-2) = (f^*(n))_{\text{Len}(f^*(n))} + (f^*(n))_{\text{Len}(f^*(n))-1}$$

Let g be defined as

$$g(x) = (x)_{\text{Len}(x)} + (x)_{\text{Len}(x)-1}$$

Then, g is primitive recursive, because $+$, $()_i$ and $\text{Len}()$ are all primitive recursive. Since $f(n) = g(f^*(n))$, f and f^* are both primitive recursive. ∎

(c) **[5 points]** The following is a proof showing that the above Fibonacci function $f(n)$ is primitive recursive:

> We prove the proposition with induction on n.
> **Base Case:** $n = 0, 1$**:** Since $f(0) = f(1) = 1$, obviously, $f(0)$ and $f(1)$ are primitive recursive.
> **Induction Step:** Suppose that $f(i)$ is primitive recursive for all $i \leq n$. Since $f(n)$ and $f(n-1)$ are primitive recursive by the induction hypothesis, and since $+$ and $-$ are also primitive recursive, $f(n+1) = f(n) + f(n-1)$ is primitive recursive. Therefore, $f(n)$ is primitive recursive.

Is this proof completely correct? If it is not completely correct, correct the mistakes to make this proof correct. If it is completely incorrect, point out the problem and elaborate why it is a problem.

Answer: This proof is completely wrong. What this proof does is proving that for every n, the function value $f(n)$ is primitive recursive. However, this does not imply that the function f itself is primitive recursive. Keep in mind that primitive recursion is defined on functions rather than on function values. ∎

4. **[15 points] (Simultaneous Recursion)** Suppose functions f_1, f_2, ..., f_k are defined as follows:

$$
\begin{aligned}
f_1(x_1, x_2, \ldots, x_m, 0) &= h_1(x_1, x_2, \ldots, x_m) \\
f_2(x_1, x_2, \ldots, x_m, 0) &= h_2(x_1, x_2, \ldots, x_m) \\
&\vdots \\
f_k(x_1, x_2, \ldots, x_m, 0) &= h_k(x_1, x_2, \ldots, x_m) \\
f_1(x_1, x_2, \ldots, x_m, t+1) &= g_1(t, f_1(x_1, x_2, \ldots, x_m, t), f_2(x_1, x_2, \ldots, x_m, t), \ldots, f_k(x_1, x_2, \ldots, x_m, t), x_1, \ldots, x_m) \\
f_2(x_1, x_2, \ldots, x_m, t+1) &= g_2(t, f_1(x_1, x_2, \ldots, x_m, t), f_2(x_1, x_2, \ldots, x_m, t), \ldots, f_k(x_1, x_2, \ldots, x_m, t), x_1, \ldots, x_m) \\
&= \\
f_k(x_1, x_2, \ldots, x_m, t+1) &= g_k(t, f_1(x_1, x_2, \ldots, x_m, t), f_2(x_1, x_2, \ldots, x_m, t), \ldots, f_k(x_1, x_2, \ldots, x_m, t), x_1, \ldots, x_m)
\end{aligned}
$$

where h_i's and g_j's are primitive recursive functions. Show that f_1, f_2, ..., f_k are all primitive recursive. (**Hint:** find and use a history function of all f_i's)

Answer: We shall provide two solutions, one elegant and the other intuitive (but complex).

Solution 1: We can encode functions f_i's with Gödel numbering. More precisely, let

$$F(x_1, \ldots, x_m, t) = [f_1(x_1, \ldots, x_m, t), f_2(x_1, \ldots, x_m, t), \ldots, f_k(x_1, \ldots, x_m, t)]$$

If we can show that F is primitive recursive, since

$$f_i(x_1, \ldots, x_m, t) = (F(x_1, \ldots, x_m, t))_i$$

f_i is also primitive recursive for every i.

Now, we can use the history function F^* of F. To show F^* and F are primitive recursive, we need to find a primitive recursive function g such that $F(t) = g(F^*(t))$. By the definition of the f_i's, we have

$$
\begin{aligned}
f_1(x_1, \ldots, x_m, t) &= g_1(t-1, f_1(x_1, \ldots, x_m, t-1), f_2(x_1, \ldots, x_m, t-1), \ldots, f_k(x_1, \ldots, x_m, t-1), x_1, \ldots, x_m) \\
f_2(x_1, \ldots, x_m, t) &= g_2(t-1, f_1(x_1, \ldots, x_m, t-1), f_2(x_1, \ldots, x_m, t-1), \ldots, f_k(x_1, \ldots, x_m, t-1), x_1, \ldots, x_m) \\
&\vdots \\
f_k(x_1, \ldots, x_m, t) &= g_k(t-1, f_1(x_1, \ldots, x_m, t-1), f_2(x_1, \ldots, x_m, t-1), \ldots, f_k(x_1, \ldots, x_m, t-1), x_1, \ldots, x_m)
\end{aligned}
$$

where $t > 0$. Since $f_i(x_1, \ldots, x_m, t-1) = (F(x_1, \ldots, x_m, t-1))_i$ and since $F(x_1, \ldots, x_m, t-1) = (F^*(x_1, \ldots, x_m, t))_t$, we have

$$F(x_1, \ldots, x_m, t) = [((F^*(x_1, \ldots, x_m, t))_t)_1, ((F^*(x_1, \ldots, x_m, t))_t)_2, \ldots, ((F^*(x_1, \ldots, x_m, t))_t)_k]$$

Note that $t = \mathrm{Len}(F^*(x_1, \ldots, x_m, t))$. Let function g be

$$g(x) = [((x)_{\mathrm{Len}(x)})_1, ((x)_{\mathrm{Len}(x)})_2, \ldots, ((x)_{\mathrm{Len}(x)})_k$$

Function g is primitive recursive since $[\,]$, $()_i$ and $\mathrm{Len}()$ are all primitive recursive. Since $F(x_1, \ldots, x_m, t) = g(F^*(x_1, \ldots, x_m, t))$, F and F^* are both primitive recursive. You can use a two-way selection to handle the case of $t = 0$. ■

Solution 2: This is a complex but naive solution. The basic idea is to incorporate all f_i's into a single function F directly and prove F is primitive recursive. Then, from F we derive the fact that f_i is primitive recursive for every i. Intuitively, function F satisfies, ignoring the x_i's, $F(0) = f_1(0)$, $F(1) = f_2(0)$, ..., $F(k-1) = f_k(0)$, $F(k) = f_1(1)$, $F(k+1) = f_2(1)$, ..., $F(2k-1) = f_k(1)$, $F(2k) = f_1(2)$, $F(2k+1) = f_2(2)$, ..., $F(3k-1) = f_k(2)$ and so on. Thus, the general definition of F is the following:

$$F(x_1, x_2, \ldots, x_k, t) = f_{\mathrm{mod}(t,k)+1}(x_1, x_2, \ldots, x_k, \lfloor t/k \rfloor)$$

To obtain f_i from F, it is simply

$$f_i(x_1, x_2, \ldots, x_k, n) = F(x_1, x_2, \ldots, x_k, k \times n + (i-1))$$

Since $f_i(n) = F(h(n))$, where $h(n) = k \times n + (i-1)$, if F is primitive, f_i is also primitive recursive for every i.

To show F is primitive primitive, we use the history function F^* of F. If we can find a primitive recursive function g such that $F(n) = g(F^*(n))$, F and F^* are both primitive recursive. Recall that the history function F^* of F is defined as follows:

$$\begin{aligned} F^*(0) &= 1 \\ F^*(n+1) &= [F(0), F(1), F(2), \ldots, F(n)] \end{aligned}$$

These F values are organized into groups, each of which has k values. Consider $F(t)$. Its position in F^* is illustrated as follows:

$$[\ldots, \underbrace{F((j-1)*k), F((j-1)*k+1), \ldots, F(j*k-1)}_{\text{group } \lfloor t/k \rfloor - 1}, \underbrace{F(j*k), F(j*k+1), \cdots \quad \overbrace{F(t)}^{\text{pos. mod}(t,k)+1} \quad , \cdots}_{\text{group } \lfloor t/k \rfloor}, \cdots]$$

From the above discussion, $F(t)$ is in position $\mathrm{mod}(t,k)+1$ of group $\lfloor t/k \rfloor$. Since $F(t)$, $t \geq k$, is computed by all values in the previous group, we have

$$F(t) = g_{\mathrm{mod}(t,k)+1}(\lfloor t/k \rfloor - 1, F((\lfloor t/k \rfloor - 1)*k), F((\lfloor t/k \rfloor - 1)*k+1), \ldots, F((\lfloor t/k \rfloor - 1)*k+(k-1)))$$

Since $F((\lfloor t/k \rfloor - 1)*k)$ is the first element of group $\lfloor t/k \rfloor - 1$, its position in F^* is $(\lfloor t/k \rfloor - 1)*k+1$, and the arguments of function $g_{\mathrm{mod}(t,k)+1}$ are $(F^*(t))_{a+1}$, $(F^*(t))_{a+2}$, ..., and $(F^*(t))_{a+k}$, where $a = (\lfloor t/k \rfloor - 1)*k$. Consequently, the relation between F and F^* is

$$F(t) = g_{\mathrm{mod}(t,k)+1}((F^*(t))_{a+1}, (F^*(t))_{a+2}, \ldots, (F^*(t))_{a+k})$$

Note that $\mathrm{Len}(F^*(t)) = t$. If we define

$$g(x) = g_{\mathrm{mod}(\mathrm{Len}(x),k)+1}((x)_{a+1}, (x)_{a+2}, \ldots, (x)_{a+k})$$

where $a = (\lfloor t/k \rfloor - 1)*k$, then we have $F(t) = g(F^*(t))$. Since g is primitive recursive (why?), F and F^* are both primitive recursive, and, all f_i's are primitive recursive. ■

5. **[15 points]** Consider the following LOOP-END instruction:

LOOP (X)	X is a variable
<*instructions*>	execute <*instructions*> X times
END	

It executes in the following way:

- Variable X is the *loop variable*. It can be any variable (*i.e.*, input, output or local)
- The *scope* of the LOOP-END instruction includes <*instructions*> and END.
- The value of the *loop variable* specifies the number of iterations the scope of a LOOP-END executes.
- However, in the course of executing a LOOP-END, the value of the *loop variable* is not changed. More precisely, the value of a loop variable can only be changed by the instructions in <*instructions*>. The following loop doubles the value of local variable Z_2:

  LOOP (Z_2)
  $\quad Z_2 \leftarrow Z_2 + 1$
  END

- The LOOP instruction can be used in a nested way. The following nested loop computes the product of X_1 and X_2:

  LOOP (X_1)
  $\quad$ LOOP (X_2)
  $\qquad Y \leftarrow Y + 1 \quad$ // add 1 to Y $X_1 \times X_2$ times
  $\quad$ END
  END

A program is called a *loop* program if and only if it is a syntactically correct program that only uses the four instructions: $V \leftarrow V$, $V \leftarrow V + 1$, $V \leftarrow V - 1$ and LOOP-END.

Show that a function is primitive recursive if and only if it is computed by a loop program. Thus, this proposition completely characterizes what primitive recursion can do. (**Hint:** Somewhere in your proof, you may need to use the simultaneous recursion theorem above).

Answer: We divide the proof into two parts as follows.

Part I: Primitive recursive functions are loop programs

In this part, we shall show that primitive recursive functions can be computed by loop programs. We shall use induction on the total number of compositions and primitive recursions that are applied to the construction a function from the initial functions.

Base Case $n = 0$: Only the initial functions do not have to be constructed using composition and primitive recursion. We show they are loop programs. Recall that the zero function is $n(x) = 0$, the successor function is $s(x) = x + 1$ and the projection function is $\pi_n^i(x_1, x_2, \ldots, x_n) = x_i$. The following shows the corresponding loop programs:

Zero	*Successor*	*Projection*
$n(x) = 0$	$s(x) = x + 1$	$\pi_n^i(x_1, x_2, \ldots, x_n) = x_i$
$Y \leftarrow 0$	$Y \leftarrow X + 1$	$Y \leftarrow X_i$

Induction Step: Suppose primitive recursive functions that are constructed with less than or equal to n compositions and primitive recursions from the initial functions can be computed by loop programs. We shall show if a primitive recursive function f that is constructed with $n + 1$ compositions and primitive recursions from the initial functions, then f can be computed by a loop program.

If the last step in the construction of f is a composition, then we have

$$f(x_1, x_2, \ldots, x_n) = h(g_1(x_1, x_2, \ldots, x_n), g_2(x_1, x_2, \ldots, x_n), \ldots, g_k(x_1, x_2, \ldots, x_n))$$

Since f is constructed from the initial functions with $n + 1$ compositions and primitive recursions, functions h, g_1, g_2, …, g_k are constructed from the initial functions in less than or equal to n compositions and primitive recursions. Therefore, by the induction hypothesis, they are computed by loop programs. For convenience, we shall use the same name for the corresponding loop program. Then, the loop program that computes f is the following:

$$\begin{array}{ll} Z_1 \leftarrow g_1(X_1, X_2, \ldots, X_n) & \text{// compute } g_1 \\ Z_2 \leftarrow g_2(X_1, X_2, \ldots, X_n) & \text{// compute } g_2 \\ \vdots & \\ Z_k \leftarrow g_k(X_1, X_2, \ldots, X_n) & \text{// compute } g_k \\ Y \leftarrow h(Z_1, Z_2, \ldots, Z_k) & \text{// compute } h \end{array}$$

If the last step in the construction of f is a primitive recursion, we have

$$\begin{aligned} f(x_1, x_2, \ldots, x_n, 0) &= g(x_1, x_2, \ldots, x_n) \\ f(x_1, x_2, \ldots, x_n, t+1) &= h(t, f(x_1, x_2, \ldots, x_n, t), x_1, x_2, \ldots, x_n) \end{aligned}$$

Functions g and h must use less than n compositions and primitive recursions, and, as a result, they are computed by loop programs. Then, the loop program for computing $f(x_1, x_2, \ldots, x_n, t)$ is the following:

$$\begin{array}{ll} Y \leftarrow g(x_1, x_2, \ldots, x_n) & \text{// initial value} \\ Z \leftarrow 0 & \text{// recursion step} \\ \text{LOOP } (T) & \text{// start the loop} \\ \quad Y \leftarrow h(Z, Y, X_1, X_2, \ldots, X_n) & \text{// recursion} \\ \quad Z \leftarrow Z + 1 & \text{// increase step} \\ \text{END} & \end{array}$$

Therefore, function f is also computed by a loop program, and the proposition holds.

Part II: Loop programs are primitive recursive

Again, we shall prove this part with induction on the depth of nesting of loops in a loop program with k variables x_1, x_2, …, x_k, one of which may be the output variable y.

Base Case $n = 0$: We shall show that loop programs with depth of nesting 0 (*i.e.*, no loop) are primitive recursive functions. Let $\mathcal{P}$ be a loop program with depth of nesting 0. Then, the statements of $\mathcal{P}$ can only be in one of the two forms: $X \leftarrow X + 1$ or $X \leftarrow X + 1$. Therefore, the output of $\mathcal{P}$ is computed using $+$ and $-$ from variables x_1, x_2, …, x_k. Because $+$ and $-$ are primitive recursive, the whole computation that yields the value in Y is also primitive recursive.

Induction Step: We assume that loop programs with depth of nesting $\leq n$ are primitive recursive functions. Consider a loop program $\mathcal{P}$ with loops of depth of nesting $n+1$. Because $\mathcal{P}$ may have an arbitrary number of loops of depth of nesting $n+1$, we shall use another level of induction on the number of loops that are of depth of nesting $n+1$.

Therefore, the "inner" induction hypothesis of this part is "for every i, a loop program has i loops that are of depth of nesting $n+1$ is primitive recursive, where $n+1$ is a fixed number."

- **Base Case** $i = 1$**:** If $\mathcal{P}$ has only one loop of depth of nesting $n+1$, it has a form of the following:

```
other non-loop statement-1
LOOP (T)
  statements, including LOOPs
  of depth of nesting n
END
other non-loop statement-2
```

Since the statements (or program) inside the LOOP-END constitute a loop program that has a depth of nesting n, it is primitive recursive. Note that program $\mathcal{P}$ uses k variables x_1, x_2, ..., x_k. The computation inside the loop can be represented with the following k functions, each of which computes the value of variable at the end of the loop:

$$
\begin{array}{l}
\text{LOOP } (T) \\
\quad x_1 = g_1(x_1, x_2, \ldots, x_n) \\
\quad x_2 = g_2(x_1, x_2, \ldots, x_n) \\
\qquad \ldots\ldots \\
\quad x_k = g_k(x_1, x_2, \ldots, x_n) \\
\text{END}
\end{array}
$$

where g_i's are primitive recursive by the induction hypothesis. For each iteration, the values of x_1, x_2, ..., x_k are computed by primitive recursive functions g_1, g_2, ..., g_k, respectively, and are used in the next iteration. Now, the loop can be translated to a primitive recursion easily as follows:

$$
\begin{array}{rcll}
f_1(x_1, x_2, \ldots, x_k, 0) & = & x_1 & \text{initial values before loop} \\
f_2(x_1, x_2, \ldots, x_k, 0) & = & x_2 & \\
 & \vdots & & \\
f_k(x_1, x_2, \ldots, x_k, 0) & = & x_k & \\
f_1(x_1, x_2, \ldots, x_k, t+1) & = & g_1(t, f_1(x_1, x_2, \ldots, x_k, t), f_1(x_1, x_2, \ldots, x_k, t), \ldots, f_k(x_1, x_2, \ldots, x_k, t)) & \\
f_2(x_1, x_2, \ldots, x_k, t+1) & = & g_2(t, f_1(x_1, x_2, \ldots, x_k, t), f_1(x_1, x_2, \ldots, x_k, t), \ldots, f_k(x_1, x_2, \ldots, x_k, t)) & \\
 & \vdots & & \\
f_k(x_1, x_2, \ldots, x_k, t+1) & = & g_k(t, f_1(x_1, x_2, \ldots, x_k, t), f_1(x_1, x_2, \ldots, x_k, t), \ldots, f_k(x_1, x_2, \ldots, x_k, t)) &
\end{array}
$$

More precisely, before entering this loop, we have the initial values of the x_i's. The loop body is rewritten into a primitive recursive form. Therefore, by the simultaneous recursion problem, all f_i's are primitive recursive functions.

Since the `other non-loop statement-1` and `other non-loop statement-2` parts are primitive recursive (base case), we know the combined program by composition is also primitive recursive. Thus, the base case holds.

- **Induction Step:** Suppose a loop program with i loop statements, each of which has a depth of nesting $n+1$, is primitive recursive. Consider a loop program $\mathcal{P}$ with $i+1$ loop statements, each of which has a depth of nesting $n+1$. We want to show that $\mathcal{P}$ is also primitive recursive.

  Program $\mathcal{P}$ can be cut into two components. The first one, $\mathcal{P}_1$, consists of every statement up to the end of the END statement of the first loop, and the second one $\mathcal{P}_2$ contains the remaining statements. Note that $\mathcal{P}_1$ and $\mathcal{P}_2$ both contain less than or equal to i loops of depth of nesting $n+1$, they are primitive recursive. Therefore, program $\mathcal{P}$ is simply the composition of the two primitive recursive functions corresponding to $\mathcal{P}_1$ and $\mathcal{P}_2$, and $\mathcal{P}$ is also primitive recursive.

In summary, a loop program with depth of nesting $n+1$ is also primitive recursive. ∎

6. **[10 points]** We shall show a *perfect debugger*, a program that can guarantee a program is bug-free, never exists. Define predicate DEBUG() as follows:

$$\text{DEBUG}(p, x) = \begin{cases} 1 & \text{if program } p \text{ running on input } x \text{ has a bug} \\ 0 & \text{otherwise} \end{cases}$$

Show that DEBUG() is not computable, and, hence, no perfect debugger is possible.

Answer: If DEBUG(p, x) is computable, we can write a program and use it in another program as a macro. Consider the following program:

$$\begin{array}{ll} \text{IF DEBUG}(X, X) \text{ GOTO } [E] & // \text{ if } x \text{ running on } x \text{ has a bug, exit} \\ Y \leftarrow \lfloor X/0 \rfloor & // \text{ otherwise, generate a bug } (e.g., \text{ division by } 0) \end{array}$$

Let the number of the above program be p. Then, we have

$$\begin{aligned} \text{DEBUG}(p, x) &\Leftrightarrow p \text{ running on } x \text{ has a bug} \\ &\Leftrightarrow \text{the line } Y \leftarrow \lfloor X/0 \rfloor \text{ is executed} \\ &\Leftrightarrow \text{DEBUG}(x, x) \text{ is false} \\ &\Leftrightarrow \neg\text{DEBUG}(x, x) \end{aligned}$$

Since x is an arbitrary input, it can be p. Therefore, substituting p for x we have

$$\text{DEBUG}(p, p) \Leftrightarrow \neg\text{DEBUG}(p, p)$$

which is absurd. Therefore, DEBUG() cannot be computable. ∎

7. **[5 points]** A function f is a constant c if and only if $f(x) = c$ for all $x \in \mathcal{N}$. The following is a proof showing that testing if a function is a constant function is not computable.

- Define a predicate CONST(p) as follows:

$$\text{CONST}(p) = \begin{cases} 1 & \text{program } p \text{ always returns } c, \text{ a constant} \\ 0 & \end{cases}$$

- Construct a program as follows:

$$\begin{array}{ll} & Y \leftarrow c + 1 \\ [A] & \text{IF CONST}(X) \text{ GOTO } [E] \\ & Y \leftarrow c \end{array}$$

  Let the program number of the above program by y.

- From the above definition of CONST(), we have

$$\begin{aligned} \text{CONST}(y) &\Leftrightarrow y \text{ is a constant function} \\ &\Leftrightarrow Y \leftarrow c \text{ always executes} \\ &\Leftrightarrow \text{CONST}(x) \text{ is false for } \textit{all } x \end{aligned}$$

  Thus, substituting y for x, the following is established:

$$\text{CONST}(y) \Leftrightarrow \neg\text{CONST}(y)$$

- Therefore, we have a contradiction and CONST() is not computable.

Is this proof completely correct? If it is not completely correct, correct the mistakes to make this proof correct. If it is completely incorrect, point out the problem and elaborate why it is a problem.

Answer: This proof is partially correct. If program y is a constant function of c, CONST(X) is always false so that program y can always return c. Thus, we have CONST(y) $\Rightarrow \neg$CONST(x). Because x is arbitrary, replacing x with y yields "CONST(y) $\Rightarrow \neg$CONST(y)." This part is correct.

On the other hand, is CONST(y) is not a constant function of c, CONST(x) is not necessary always true. In other words, if some x may cause CONST(x) to be true and some other x may cause CONST(x) to be false, program y is not a constant function of c. Thus, CONST(x) being false for some x does not imply CONST(y) being false (*i.e.*, y being a constant function of c). Consequently, the above is partially correct. A better construction for y is required. ■

Your Problems

- When you provide a formula and claim it is primitive recursive, you have to supply a detailed proof or a convincing argument about the correctness of your result. If you only write down a formula and claim that it satisfies some conditions without a proof in all subsequent homework and exams, you will receive no point for that problem. Since this has been mentioned on the web page and at the very beginning of this semester, I will enforce this rule without any leniency. "A proof being too long" is **not** an excuse.
- Similar to the above, some answered Problem 6 in the following way:
  1. Construct a program from DEBUG(X,X).
  2. Write down something like the following without any discussion:

     $$\text{DEBUG}(p, x) \Leftrightarrow \neg\text{DEBUG}(x, x)$$

  3. Therefore, DEBUG() is not computable.

  I expect a discussion that can establish the fact DEBUG$(p, x) \Leftrightarrow \neg$DEBUG$(x, x)$. Should this happen again, you will receive no point for this type or answer.