Nelson Batista

CSC 33500 Programming Language Paradigms — Section R

9/20/16

Homework 2

Abelson and Sussman

1.11

$$f(n) = \begin{cases} n & n < 3 \\ f(n-1) + 2f(n-2) + 3f(n-3) & n \geq 3 \end{cases}$$

A recursive solution is given below:

```
(define (function-recur n)
        (cond ((< n 3) n)
              (else (+ (function-recur (- n 1))
                       (* 2 (function-recur (- n 2)))
                       (* 3 (function-recur (- n 3)))))))
```

Proof:

We first see that, for any value $n < 3$, `function-recur n` returns n, due to the first `cond` expression. For our induction hypothesis, we assume that, for some arbitrary value k, `function-recur k` correctly returns $f(k-1) + 2f(k-2) + 3f(k-3)$. The recursive calls in the `cond`'s `else` expression exhibit exactly this behavior. Our goal is thus to show that `function-recur` will return the correct value for $k+1$, or that $f(k+1) = f(k) + 2f(k-1) + 3f(k-2)$. We have:

$$f(k+1) = f((k+1)-1) + 2f((k+1)-2) + 3f((k+1)-3)$$

$$f(k+1) = f(k) + 2f(k-1) + 3f(k-2)$$

The last statement shows that, assuming `function-recur k` returns the correct value, so will `function-recur (+ k 1)`.

An iterative solution is given below:

```
(define (function-iter n)
```

```
  (cond ((< n 3) n)
  (else (function-tail n 2 2 1 0))))


(define (function-tail n count result a b)
  (cond ((= count n) result)
        (else (function-tail n (+ count 1)
                             (+ result (* 2 a) (* 3 b)) result a))))
```

Proof:

We see that the first line of the function will have it return n in the event that $n <= 3$. That behavior is thus fulfilled. When $n \neq 3$, we have a call to a "helper" function that handles the iteration. This function, `function-tail` is tail-recursive. The `result` parameter holds the current result of the function call, which is accurate if we take n to be the current value of `count` at any given point in the computation. This is guaranteed at all points of the iteration. When `function-tail` is first called, it is passed the arguments `n = n, count = 2, result = 2, a = 1, b = 0`. We see the initial value of `result` is indeed the correct value of $f(\texttt{count})$, 2. The function iterates, and at each step, the values of $f(\texttt{count} - 2)$ and $f(\texttt{count} - 3)$ are stored and updated in `a` and `b`, respectively. At `count = 3`, the value of `result` is `2 + 2*(1) + 3*(0)`, which is 4 and therefore the correct value for $f(3)$. If $n \neq 3$, then the function is called again with an incremented value of `count`, and `a` is replaced with the current value of `result` to remain consistent with our observation that `a` is the value of $f(n-1)$, and `b` is replaced with the current value of `a` to contain the value of $f(n-2)$.

This chain continues upward until $\texttt{count} = n$, at which point the function will return the current value of `result`, which we have demonstrated will contain the correct value of $f(n)$.

1.12

A recursive solution to calculate an element of Pascal's triangle given a particular row and position within the row is given below:

```
(define (pascal row num)
        (cond ((or (<= num 1) (>= num row)) 1)
              (else (+ (pascal (- row 1) (- num 1))
                       (pascal (- row 1) num)))))
```

Proof:

Note that a particular element in a particular row of Pascal's triangle can be found using

$$f(row, elem) = \begin{cases} 1 & row <= 2 \text{ or } elem = 1 \text{ or } elem = row \\ f(row-1, elem-1) + f(row-1, elem) & otherwise \end{cases}$$

First we observe that the function returns the correct value for any element along the edges of the triangle. If the element asked for within a given row is at position 1, or at the last position in the row (a given row n in Pascal's triangle has n elements in it), the function returns 1, due to the first expression in the `cond` statement. This is correct behavior, as even the question itself notes that any element on the edge of a row will be 1.

To prove that the function works for all elements, we first assume it works for any arbitrary element k in any arbitrary row r. We have already demonstrated that, no matter what value we set for r, the function is correct if k happens to be the first or last element in the row. Beyond that, we must show that, assuming `pascal r k` correctly returns `(+ (pascal (- r 1) (- k 1)) (pascal (- r 1) k))`, it will also return the correct value for $r+1$ and $k+1$. That is, `pascal (+ r 1) (+ k 1)` should return the sum of `pascal r k` and `pascal r (+ k 1)`, the sum of the element before k in the row above r and the element at position $k+1$ in the row above $r+1$.

```
pascal (+ r 1) (+ k 1) = (+ (pascal (- (+ r 1) 1) (- (+ k 1) 1))
                            (pascal (- (+ r 1) 1) (+ k 1)))
  pascal (+ r 1) (+ k 1) = (+ (pascal r k) (pascal r (+ k 1)))
```

Thus, `pascal (+ r 1) (+ k 1)` returns the correct value for some arbitrary element k in some arbitrary row r.

Sum of digits

A recursive solution to return the sum of digits in a non-negative integer is below:

```
(define (sum-digits-recur num)
        (cond ((<= num 0) 0)
              ((< num 10) num)
              (else (+ (sum-digits-recur (quotient num 10))
                       (modulo num 10)))))
```

Proof:

First, we note that, if a number n is less than 10, the sum of its digits is simply n, since it has only that digit. This is satisfied by the second expression in the `cond` statement, with the first expression covering the precondition that our input will not be a negative integer. We then assume that the function returns the correct sum of the digits of an arbitrary number k, equal to the sum of `(sum-of-digits-recur (quotient k 10))` and `(modulo k 10)`. The function will be called with progressively fewer digits until only a single one remains, at which point it will return that digit's value and travel up the stack, adding the rest of the digits through the deferred addition. We assume `(sum-of-digits-recur k)` = `(+ (sum-of-digits-recur (quotient k 10)) (modulo k 10)`. We must then prove `(sum-of-digits-recur (+ k 1)` = `(+ (sum-of-digits-recur (quotient (+ k 1) 10)) (modulo (+ k 1) 10)`. This is a trivial substitution.

The iterative solution:

```
(define (sum-digits-iter num)
        (sum-digits-tail num num))


(define (sum-digits-tail num sum)
        (cond ((< num 10) sum))
              (else (sum-digits-tail (quotient num 10)
                                     (+ sum (modulo num 10)))))
```

Increasing order of digits

A recursive solution to return whether a given number's digits are in increasing order is given below:

```
(define (increasing-recur num)
  (cond ((< num 100) (>= (modulo num 10) (quotient num 10)))
        (else (and (>= (modulo num 10) (modulo (quotient num 10) 10))
                   (increasing-recur (quotient num 10))))))
```

An iterative solution:

```
(define (increasing-iter num)
        (increasing-tail num 9))


(define (increasing-tail num digit)
        (cond ((> (modulo num 10) digit) #f)
              ((< num 10) #t)
              (else (increasing-tail (quotient num 10)
```

```
                        (modulo num 10)))))
```