Exam I — Sample Solutions

Problem 1.
Notice that the structure of the derivation tree reflects the precedence of the guards over the result evaluation by placing them at a lower level.

Problem 2.
This problem involves the analysis and comparison of three Haskell function definitions (a, b, and c) with three BNF definitions.

Function a corresponds to (iii). This is seen by noticing that a [ ] = False, in agreement with the null string not being in the language of (iii) since there is always at least one '1'. For longer strings \(0^k1w\) (where \(w\) is any string) in the language of (iii), \(a 0^k1w = a 0^{k-1}1w = \ldots = a 1w = True\) as required. This leaves only strings of the form \(0^*\) which are not in the language (iii), and \(a 0^k = False\) as required.

Function b corresponds to (i). An inductive analysis is used for this part. First of all, notice that the function b is consistent with (i) since the null string is in this language, and no string of length 1 is. This is the basis for the induction. Next we assume that function b is consistent with (i) for all strings of length n or less, and consider a string x of length \(n+1\) \((n \geq 1)\). There are two cases based on the second bit of x —

- **Case 1:** \(x = 0y\) for \([0,1)\) and \(y([0,1]^\ast)\). In this case, if \(x \in L(i)\), then it must be formed using an initial choice from \(0^*\), namely \(0^k1\) where \(k \geq 1\), \(y = 0^{k-1}1y'\), and \(y'\) is formed by the remaining choices so \(y' \in L(i)\) and so by the inductive hypothesis, \(b y' = True\). But \(b x = b \[0^{k-1}1y' = b \[0^{k-2}1y' = \ldots = b \[1y' = b y'\) extending the induction.
Case 2: $x = \{0,1\}^1 y$ for $\{0,1\}$ and $y \in \{0,1\}^*$. In this case, if $x \in L(i)$, then it must be formed using an initial choice from $\{0,1\}^*$, namely $\{1\}$. But then $y$ is formed by the remaining choices so $y \in L(i)$ and so by the inductive hypothesis, $b y = True$. But $b x = b y$ in this case extending the induction.

Function $c$ corresponds to (ii). This function yields True for palindromes. BNF $X$ also derives exactly the palindromes since at each non-terminating step the same letter is introduced at the front and back of the string (e.g., $X \rightarrow 0X0 \rightarrow 01X10 \rightarrow 01010$) so when the derivation terminates the result is a palindrome.

Problem 3.
Writing this program requires a repetition determined by the length of the coefficient list, and can be approached either iteratively or recursively. List comprehension provides a natural technique to write this program. Since the index of a coefficient and the power of the argument $x$ agree in each term of the polynomial, this index can be used to create the polynomial term sequence — for $as!!k$, the power is $x^k$.

$$\text{evalPoly as } x = \text{sum } [(as!!k) \times x^k | k\leftarrow[0..\text{length } as-1]]$$

Alternatively, a basic recursion reducing the length of the coefficient list can be used. In this approach, the length of the initial coefficient list gives the exponent for the highest power (i.e., last) term in the polynomial expansion, and the evaluation proceeds “from right to left” through smaller and smaller powers. Namely,

$$\text{evalPoly \ [a] x = a } \quad \quad \text{-- one coefficient so it’s a constant polynomial}$$
$$\text{evalPoly as } x = (\text{last } as) \times x^\text{(length as –1)} + \text{evalPoly (init as) } x$$

A more sophisticated “forward recursion” that develops the powers implicitly is based on the identities

$$a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1} = a_0 + x^* (a_1 + a_2 x + \ldots + a_{n-1} x^{n-2}) = a_0 + x^* (a_1 + x^* (a_2 + a_3 x + \ldots + a_{n-1} x^{n-3})) = \ldots$$

This analysis gives the recursion

$$\text{evalPoly \ [a] x = a}$$
$$\text{evalPoly \ (a:as) x = a + x^* evalPoly as x}$$