CS 337 **Solution to Test 3** **5/10/07**

1. (**PART 3**: Proofs of Recursive Programs) Proof is by induction on n in the definition of `grayGen`.

   - $n = 0$: `rev [""] = [""]`

   - $n > 0$: we have to show `rev((cons0 a) ++ (cons1 b)) = (cons1 a) ++ (cons0 b)`

$$
\begin{array}{ll}
 & rev((cons0\ a)\ ++\ (cons1\ b)) \\
= & \{rev(xs\ ++\ ys) = (rev\ ys)\ ++\ (rev\ xs)\} \\
 & rev(cons1\ b)\ ++\ rev(cons0\ a) \\
= & \{rev(cons1\ ys) = cons1(rev\ ys)\} \\
 & cons1(rev\ b)\ ++\ rev(cons0\ a) \\
= & \{rev(cons0\ ys) = cons0(rev\ ys)\} \\
 & cons1(rev\ b)\ ++\ cons0(rev\ a) \\
= & \{\text{induction: } rev\ a = b \text{ where } (a,b) = grayGen\ n \\
 & \quad \text{Also, } rev(rev\ a) = (rev\ b), \text{ or } a = (rev\ b) \text{ using } rev(rev\ xs) = xs\} \\
 & (cons1\ a)\ ++\ (cons0\ b)
\end{array}
$$

2. (**PART 3**: Higher Order Functions)

   (a)
   ```
   zip ([], []) = []
   zip ((x:xs), (y: ys)) = (x,y): zip (xs,ys)
   zip :: ([a],[b]) -> [(a,b)]
   ```

   (b)
   ```
   unzip [] = ([], [])
   unzip ((x,y): xyss) = ((x:xs), (y: ys))
                         where (xs, ys) = unzip xyss
   unzip :: [(a,b)] -> ([a],[b])
   ```

   (c)
   ```
   cross (f,g) (x,y) = (f x , g y)
   cross :: (a -> b,c -> d) -> (a,c) -> (b,d)
   ```

3. (**PART 3**: Rabin-Karp String Matching; 11 points)

   (a) Treat the strings as binary. Successful matches are shown with a bar over the string: $0\overline{1101}0001\overline{1101}\ \overline{1101}$.
   With $q = 3$ and 1101 mod 3 = 1, we get possible collisions at x provided x mod 3 = 1 and x represents a 4-bit string. That is, collisions occur at 0001(1), 0100(4), 0111(7) and 1010(10). I show the collisions in three groups (this is due to the limitation of my math editing software):
   $01\overline{1010}0011\overline{0111}01$, and
   $0110\overline{1000}11011101$, and
   $01101\overline{0001}1011101$

The collisions do decrease with $q = 5$, but no general inference can be drawn from it. Since 1101 mod 5 = 3, we expect collisions at 0011(3) and 1000(8). I show the collisions in two groups, as before:
$0110\overline{1000}11011101$, and
$0110\overline{10001}1011101$

(b) Use a single q for all patterns; let p_i be derived by applying mod q to the i^{th} pattern.

Nothing special needs to be done if the patterns are of the same length. Use the same algorithm as before to go over the text, and for each symbol in the text check against all p_is.

For the general case of patterns of arbitrary lengths, let m be the maximum pattern length. After the initial prefix of length m in the text, each symbol in the text yields a value which is again matched against all p_is. In the initial prefix of length m, only patterns of the appropriate length are matched.

4. (**PART 3**: String Matching)

(a) The core function is monotonic, that is,

$$
\begin{array}{ll}
u \preceq v \;\Rightarrow\; c(u) \preceq c(v) & \\
\qquad u \preceq v & \\
\Rightarrow & \{\text{given } c(u) \prec u \text{ and } \preceq \text{ a partial order}\} \\
\qquad c(u) \prec v & \\
\Rightarrow & \{\text{definition of core: } u \preceq c(v) \;\equiv\; u \prec v. \\
& \quad \text{Substitute } c(u) \text{ for } u \text{ and } v \text{ for } v \text{ in the right side}\} \\
\qquad c(u) \preceq c(v) &
\end{array}
$$

(b) Given that $u \preceq v$, it is not necessarily true that $us \preceq vs$, for any symbol s. Consider $u = a$ and $v = aba$. Then, $u \preceq v$. However, $as \preceq abas$ does not hold unless $s = b$.

(c) The core computation does not have to be modified.The only modification to the algorithm in P. 163 is to replace the condition $t[r] = p[r - l]$ by $t[r] = * \vee t[r] = p[r - l]$ (and, similarly, $t[r] \neq p[r - l]$ by $t[r] \neq * \wedge t[r] \neq p[r - l]$).

$$
\begin{array}{ll}
t[r] = * \vee t[r] = p[r - l] & \longrightarrow\; r := r + 1 \\
\quad \{ \text{ more text has been matched } \} & \\
t[r] \neq * \wedge t[r] \neq p[r - l] \;\wedge\; r = l & \longrightarrow\; l := l + 1;\; r := r + 1 \\
\quad \{ \text{ we have an empty string matched so far;} & \\
\quad\quad \text{the first pattern symbol differs from the next text symbol } \} & \\
t[r] \neq * \wedge t[r] \neq p[r - l] \;\wedge\; r > l & \longrightarrow\; l := l' \\
\quad \{ \text{ a nonempty prefix of } p \text{ has matched but the next symbols don't } \} &
\end{array}
$$

5. (**PART 3**: Relational Databases)

   (a) List of workers and their managers, where the worker salary is below $20,000.

   $$\pi_{worker,\ manager}(\sigma_{salary<20000}(M \bowtie S))$$

   (b) List of workers whose spouse is their manager.

   $$\pi_{worker}(\sigma_{manager=spouse}(F \bowtie M))$$

   (c) Average salary by department.

   $$_{Dept}\mathbf{A}_{Avg\ salary}(M \bowtie S)$$

   (d) Workers who are paid more than their managers.

   $$\pi_{worker}(\sigma_{salary>man\_salary}(M \bowtie S \bowtie S_{worker:=manager;salary:=man\_salary}))$$

Solutions to bonus questions are on the next page

Solutions to bonus questions

6. (**PART 1**: Error Correction)

   (a) Every word at distance 4 from a codeword is not itself a codeword. 1 1 1 1 1 1 1 1 is a codeword (top row of Table 2.12). But 0 0 0 0 1 1 1 1 is not.

   (b) If the sender sends 1 0 0 1 1 0 0 1 and the receiver receives 1 1 1 1 1 0 1 1, the Hamming distance between the two words is 3. Since the distance among codewords is exactly 4, the received message is not a codeword; so the receiver can detect the error.

   He will pick the closest codeword to 1 1 1 1 1 0 1 1 which is 1 1 1 1 1 1 1 1.

   (c) No. Suppose 1 1 1 1 1 1 1 1 is sent and it is corrupted to 1 0 1 0 1 0 1 0; there is exactly two errors in each half, left and right. The received word is also a codeword. So the receiver can not tell if 1 0 1 0 1 0 1 0 was sent and received perfectly, or 1 1 1 1 1 1 1 1 was sent and received erroneously.

7. (**PART 2**: Finite State Machine)

   (a) A machine that accepts exactly half the binary strings: the machine rejects all strings starting with 0 and accepts all starting with 1.

   (b) A machine that accepts a binary string unless 111 is a substring.

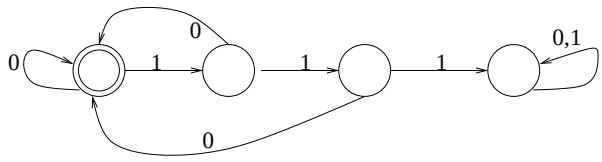
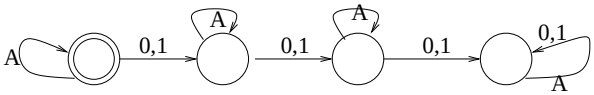


Figure 1: Accept unless 111 is a substring.

   (c) A machine that receives a string of bit-pairs (x, y) as input and accepts if there are at least 3 inputs pairs (x, y) where $x < y$.



A= {(0,0), (1,0), (1,1)}