CS 337 **Solution to Test 1** **4/6/09**

1. (Finite State Machine Design)

   (a) See Figure 1.

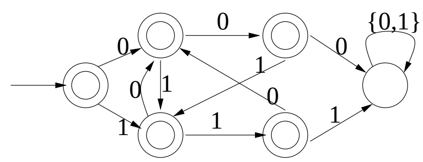

Figure 1: Binary string without 3 consecutive identical symbols

   (b) See Figure 2.

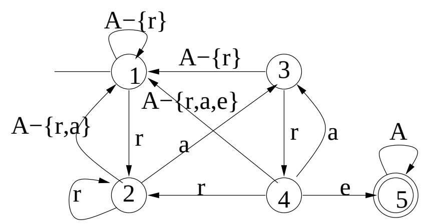

Figure 2: accept if input contains "rare"

2. (Reasoning about Finite State Machines)

   (a) See Figure 3.

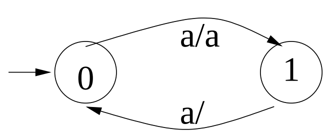

Figure 3: output every other symbol

   (b) For arbitrary symbols a and b, and string x

$$f(\epsilon) = \epsilon,\ f(a) = a,\ f(abx) = af(x)$$

   (c) Let $evenl(x)$ denote that the length of x is even. Associate predicates p_0 and p_1 with states 0 and 1, where

$$p_0 \equiv evenl(x) \wedge y = f(x)$$
$$p_1 \equiv \neg evenl(x) \wedge y = f(x)$$

(d) $evenl(\epsilon) \ \wedge \ \epsilon = f(\epsilon)$
$\neg evenl(x) \ \wedge \ y = f(x) \ \Rightarrow \ evenl(xa) \ \wedge \ y = f(xa)$, for all a
$evenl(x) \ \wedge \ y = f(x) \ \Rightarrow \ \neg evenl(xa) \ \wedge \ ya = f(xa)$, for all a

3. (Writing Recursive Programs)

(a)
```
grade [] = ([],[],[])
grade((name,score): xs)
 | score >= 90 = ((name:a),b, c)
 | score >= 80 = (a,(name:b), c)
 | otherwise   = (a,b, (name:c))
    where (a,b,c) = grade xs
```

(b)
```
suffix [] = [[]]
suffix (x:xs) = (x:xs):(suffix xs)
```

(c)
```
cart1 x [] = []
cart1 x (y:ys) = (x,y) : (cart1 x ys)

cart [] ys = []
cart (x:xs) ys = (cart1 x ys) ++ (cart xs ys)
```

(d) We define function `scan` that has three arguments: (1) the part of the string that has already been scanned, call it `left`, (2) the left paren count − the right paren count over `left`, call it `n`, and (3) the part of the string that remains to be scanned, call it `right`. Function `scan` returns True iff `left ++ right` is balanced.
Then, `balanced xs = scan [] 0 xs`.

In defining `scan` we will ensure that `n` is non-negative. The function is easy to write:

```
scan left n ""       = n == 0
scan left 0 (')':xs) = False
scan left n (')':xs) = scan (left ++ ")") (n-1) xs
scan left n ('(':xs) = scan (left ++ "(") (n+1) xs
```

Now observe that `left` is used only in computing its own next value; it does not affect the other two arguments in the last two clauses, nor the result in the first two clauses. So, we can eliminate `left` altogether.

```
scan n ""       = n == 0
scan 0 (')':xs) = False
scan n (')':xs) = scan (n-1) xs
scan n ('(':xs) = scan (n+1) xs
```

Then,

```
balanced xs = scan 0 xs
```

4. (Properties of Recursive programs) The proof of `rr(lr xs) = xs` is by case discrimination, `xs = []` and `xs` $\neq$ `[]`. Note that we never employ an inductive hypothesis; all induction are buried in the given facts (0–3).

- `xs = []`:

We have to show: `rr(lr []) = []`

```
     rr(lr [])
=  {lr [] = []}
     rr []
=  {rr [] = []}
     []
```

- Input list is non-empty:

We have to show `rr(lr (x:xs)) = (x:xs)`

```
     rr(lr (x:xs))
=  {from definition of lr, lr (x:xs) = xs ++ [x] }
     rr(xs ++ [x])
=  {definition of rr}
     y:(rev ys) where y:ys = rev (xs ++ [x])
=  {from given fact (2): rev (xs ++ [x]) = (rev [x]) ++ (rev xs)}
     y:(rev ys) where y:ys = (rev [x]) ++ (rev xs)
=  {from given fact (0): rev [x] = [x]}
     y:(rev ys) where y:ys = [x] ++ (rev xs)
=  {from given fact (3): [x] ++ (rev xs) = x:(rev xs)}
     y:(rev ys) where y:ys = x:(rev xs)
=  {substituting for y and ys}
     x:(rev(rev xs))
=  {from given fact (1): rev(rev xs) = xs}
     x:xs
```