CPSC 201: Introduction to Computer Science
Carsten Schürmann
Date: February 4, 2002

Solutions 1

Guidelines

While we acknowledge that beauty is in the eye of the beholder, you should nonetheless strive for elegance in your code. Not every program which runs deserves full credit. Make sure to state invariants in comments which are sometimes implicit in the informal presentation of an exercise. If auxiliary functions are required, describe concisely what they implement. Do not reinvent wheels, and try to make your functions small and easy to understand. Use tasteful layout and avoid long winded and contorted code. None of the problems requires more than a few lines of SML code.

Problem 1: Poker (25 points)

Recall the Poker example from class (see also the webpage). In this example, we have seen three functions that determine if a given hand of cards contains a pair, three of a kind, or four of a kind. For poker, however, we also need to be able to decide if a hand contains

- a flush (five cards of the same suit),

```
(* has' : (Suit -> bool) -> bool
   has' P = B

   Invariant:
   If P is a predicate
   B holds iff P holds for at least one suit S
*)
fun has' P =
    P Heart orelse P Diamond orelse P Spade orelse P Club

(* flush : Hand -> bool
   flush H = B

   Invariant:
   B holds iff hand H contains a flush
*)
fun flush H = has' (fn S => cards_of_a_suit 5 S H)
```

- a straight (five cards of arbitrary suit, but increasing rank),

```
(* has'' : (Rank -> bool) -> bool
   has'' P = B

   Invariant:
   If P is a predicate
   B holds iff P holds for at least one ranks Two <= R <= Ten
*)
fun has'' P =
    P Ten orelse P Nine orelse P Eight orelse P Seven orelse
    P Six orelse P Five orelse P Four orelse P Three orelse P Two

(* series : int -> Rank -> Hand -> bool
   series n R H = B

   Invariant:
   B holds iff H contains a R low series of n cards
*)
fun series 0 H R = true
  | series 1 H Ace = exists H Ace
  | series n H R = exists H R andalso series (n-1) H (next R)

(* straight :  Hand -> bool
   straight H = B

   Invariant:
   B holds iff H contains a straight
*)
fun straight H = has'' (fn R => series 5 H R)
```

- a straight flush (five card of the same suit and increasing rank).

```
(* straightflush :  Hand -> bool
   straightflush H = B

   Invariant:
   B holds iff H contains a straight flush
*)
fun straightflush H = straight H andalso flush H
```

Give the type for each of the functions, and program the function in ML, using the code accessible through the webpage (Lecture 4).

Problem 2: Conversion of integers (25 points)

1. Write a function

```
int_to_roman : int -> RomanNumber
```

that converts integers between 1 and 999 to their Roman numeral equivalent. Roman numbers should be elements of the following datatype

```
datatype RomanDigit
  = I | V | X | D | L | C | M
datatype RomanNumber
  = Empty
  | Cons of RomanDigit * RomanNumber
```

The following chart summarizes how integer digits are written as Roman numerals are written.

	1	2	3	4	5	6	7	8	9
ones	I	II	III	IV	V	VI	VII	VIII	IX
tens	X	XX	XXX	XL	L	LX	LXX	LXXX	XC
hundreds	C	CC	CCC	CD	D	DC	DCC	DCCC	CM

Let's use the number 843 as an example of how the conversion takes place. First, we look up the hundreds digits (8) in the hundreds row and find DCCC. Second we look up the thens digits (4) and the ones digit (3) and find XL and III, respectively. Finally, we concatenate the three together to form the final Roman numeral: DCCCXLIII.

Hint: To create the function, break the problem into subproblems (simpler functions). One particular way of partitioning this problem is given below. Since this is not the only good partitioning, you are encouraged to come up with your own solution.

```
 hundreds      : int -> RomanNumber
 tens          : int -> RomanNumber
 ones          : int -> RomanNumber
 int_to_roman  : int -> RomanNumber

(* ones : int -> RomanNumber
   ones i = N

   Invariant:
   If  0 <= i <= 9 then
   N is the roman numeral that corresponds to i
*)
fun ones 0 = Empty
  | ones 1 = Cons (I, Empty)
  | ones 2 = Cons (I, Cons (I, Empty))
  | ones 3 = Cons (I, Cons (I, Cons (I, Empty)))
  | ones 4 = Cons (I, Cons (V, Empty))
  | ones 5 = Cons (V, Empty)
  | ones 6 = Cons (V, Cons (I, Empty))
  | ones 7 = Cons (V, Cons (I, Cons (I, Empty)))
```

```
  | ones 8 = Cons (V, Cons (I, Cons (I, Cons (I, Empty))))
  | ones 9 = Cons (I, Cons (X, Empty))

(* tens : int -> RomanNumber -> RomanNumber
   ones i N' = N

   Invariant:
   If  0 <= i <= 9 then
   If   N' is the Roman Numeral that corresponds to j
   then N is the Roman Numeral that corresponds to 10*i+j
*)
fun tens 0 N = N
  | tens 1 N = Cons (X, N)
  | tens 2 N = Cons (X, Cons (X, N))
  | tens 3 N = Cons (X, Cons (X, Cons (X, N)))
  | tens 4 N = Cons (X, Cons (L, N))
  | tens 5 N = Cons (L, N)
  | tens 6 N = Cons (L, Cons (X, N))
  | tens 7 N = Cons (L, Cons (X, Cons (X, N)))
  | tens 8 N = Cons (L, Cons (X, Cons (X, Cons (X, N))))
  | tens 9 N = Cons (X, Cons (C, N))

(* hundreds : int -> RomanNumber -> RomanNumber
   ones i N' = N

   Invariant:
   If   0 <= i <= 9 then
   If   N' is the Roman Numeral that corresponds to j
   then N is the Roman Numeral that corresponds to 100*i+j
*)
fun hundreds 0 N = N
  | hundreds 1 N = Cons (C, N)
  | hundreds 2 N = Cons (C, Cons (C, N))
  | hundreds 3 N = Cons (C, Cons (C, Cons (C, N)))
  | hundreds 4 N = Cons (C, Cons (D, N))
  | hundreds 5 N = Cons (D, N)
  | hundreds 6 N = Cons (D, Cons (C, N))
  | hundreds 7 N = Cons (D, Cons (C, Cons (C, N)))
  | hundreds 8 N = Cons (D, Cons (C, Cons (C, Cons (C, N))))
  | hundreds 9 N = Cons (C, Cons (M, N))

(* int_to_roman : int -> RomanNumber
   int_to_roman i = N

   Invariant:
   If  0 <= i <= 999 then
   N is the Roman Numeral that corresponds to i
```

```
*)
fun int_to_roman n =
    let
      val on = n mod 10
      val on' = n div 10
      val te = on' mod 10
      val te' = on' div 10
      val hu = n mod 10
    in
      hundreds hu (tens te (ones on))
    end
```

2. Write a function that does the inverse of what `int_to_roman` does:

```
roman_to_int : RomanNumber -> int
```

```
(* ones' : RomanNumber -> int
   ones' N = i

   Invariant:
   If  <= N <= IX then
   i is the integer that corresponds to N
*)
fun ones' (Cons (I, Cons (X, Empty))) = 9
  | ones' (Cons (V, Cons (I, Cons (I, Cons (I, Empty))))) = 8
  | ones' (Cons (V, Cons (I, Cons (I, Empty)))) = 7
  | ones' (Cons (V, Cons (I, Empty))) = 6
  | ones' (Cons (V, Empty)) = 5
  | ones' (Cons (I, Cons (V, Empty))) = 4
  | ones' (Cons (I, Cons (I, Cons (I, Empty)))) = 3
  | ones' (Cons (I, Cons (I, Empty))) = 2
  | ones' (Cons (I, Empty)) = 1
  | ones' (Empty) = 0

(* tens' : RomanNumber -> int
   tens' N = i

   Invariant:
   If  <= N <= XCIX then
   i is the integer that corresponds to N
*)
fun tens' (Cons (X, Cons (C, N))) = 90 + ones' N
  | tens' (Cons (L, Cons (X, Cons (X, Cons (X, N))))) = 80 + ones' N
  | tens' (Cons (L, Cons (X, Cons (X, N)))) = 70 + ones' N
  | tens' (Cons (L, Cons (X, N))) = 60 + ones' N
  | tens' (Cons (L, N)) = 50 + ones' N
```

```
  | tens' (Cons (X, Cons (L, N))) = 40 + ones' N
  | tens' (Cons (X, Cons (X, Cons (X, N)))) = 30 + ones' N
  | tens' (Cons (X, Cons (X, N))) = 20 + ones' N
  | tens' (Cons (X, N)) = 10 + ones' N
  | tens' N = ones' N

(* hundreds' : RomanNumber -> int
   hundreds' N = i

   Invariant:
   If  <= N <= CMXCIX then
   i is the integer that corresponds to N
*)
fun hundreds' (Cons (C, Cons (M, N))) = 900 + tens' N
  | hundreds' (Cons (D, Cons (C, Cons (C, Cons (C, N))))) = 800 + tens' N
  | hundreds' (Cons (D, Cons (C, Cons (C, N)))) = 700 + tens' N
  | hundreds' (Cons (D, Cons (C, N))) = 600 + tens' N
  | hundreds' (Cons (D, N)) = 500 + tens' N
  | hundreds' (Cons (C, Cons (D, N))) = 400 + tens' N
  | hundreds' (Cons (C, Cons (C, Cons (C, N))))  = 300 + tens' N
  | hundreds' (Cons (C, Cons (C, N))) = 200 + tens' N
  | hundreds' (Cons (C, N)) = 100 + tens' N
  | hundreds' N = tens' N

(* roman_to_int : RomanNumber -> int
   roman_to_int N = i

   Invariant:
   If  <= N <= CMXCIX then
   i is the integer that corresponds to N
*)
fun roman_to_int N = hundreds' N
```

3. Write a function

```
int_to_string : int -> string
```

that converts an integer to its representation as a string. For example, `int_to_string(27)` should return the string `"27"`. Make sure your function handles all integers, not just the positives (though you may want to write a helper function specifically for positive integers).

Though you are not to use SML/NJ's built-in `makestring` function, you should check that your function gives the same results.

```
(* int_to_string' : int -> string
```

```
   int_to_string' i = s

   Invariant:
   If   i>0
   s is the string that corresponds to i
*)
fun int_to_string' 0 = ""
  | int_to_string' n =
    let
      val (k, r) = (n div 10, n mod 10)
    in
      (int_to_string' k) ^ (Int.toString r)
    end

(* int_to_string : int -> string
   int_to_string i = s

   Invariant:
   s is the string that corresponds to i
*)
fun int_to_string 0 = "0"
  | int_to_string n =
    if n > 0 then int_to_string' n
    else "~" ^ int_to_string' (~n)
```

Problem 3: Binomial numbers (50 points)

Binomial numbers play an important role in combinatorics because they answer the question of how many subsets can be formed by picking r elements out of a set of n (n choose r, or n over r). There are several ways to calculate binomials. One way is by using factorials, the other one by reading them out of Pascal's triangle (explained below). One of the aims of this problem is to show the equivalence between both definitions.

First, the factorial function on natural numbers is defined mathematically as follows:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{if } n > 0 \end{cases}$$

The factorial function is the basis for the first definition of binomial numbers. We write $\binom{n}{r}$ for binomial numbers that are defined as follows

$$\binom{n}{r} = \frac{n!}{r!(n-r)!} \qquad \text{for } 0 \leq r \leq n.$$

A way to calculate binomial numbers which does not require factorials is to use Pascal's triangle. The triangle has a 1 at its top, every other number in the Pascal triangle is defined as the sum of the numbers to the right and to the left above. If there isn't any number, it counts as 0.

$$
\begin{array}{ccccccc}
 & & & 1 & & & n=0 \\
 & & 1 & & 1 & & n=1 \\
 & 1 & & 2 & & 1 & n=2 \\
1 & & 3 & & 3 & & 1 \quad n=3 \\
1 & & 4 & & 6 & & 4 \quad 1 \quad n=4
\end{array}
$$

1. Write a function

   ```
   pascal : int * int -> int
   ```

   that computes numbers following Pascal's triangle. Calculate the following values of `pascal` if applied to `(5,3)` and `(7,1)`.

   ```
   (* pascal : int * int -> int
      pascal (n, r) = m

      Invariant:
      m = n over r
   *)
   fun pascal (n, 0) = 1
     | pascal (n, r) =
       if n=r then 1
       else pascal (n-1, r-1) + pascal (n-1, r)
   ```

   `pascal (5,3)` $\Downarrow 10$ and `pascal (7,1)` $\Downarrow 7$.

2. Show by induction that your implementation of `pascal` computes binomial numbers. That is, you should prove that for any $n, r \geq 0$ natural numbers

   $$\texttt{pascal (n, r)} \Downarrow \binom{n}{r}$$

   Hint: Choose first the induction variable. Then prove the base case, and finally the successor case. State carefully which additional assumptions you can use in the induction step.

   Case: $n = 0$. `pascal (0, r)` $\Downarrow 1 = \binom{0}{r}$.

   Case: $n = m + 1$.

   Case: $r = n$. `pascal (n, n)` $\Downarrow 1 = \binom{n}{n}$.

   Case: $0 \leq r \leq n$.

   ```
   pascal (n, r)
   ⇒ pascal (m, r-1) + pascal (m, r)
   ```

$$
\begin{aligned}
&\Rightarrow \binom{m}{r-1} + \binom{m}{r} \\
&= \frac{m!}{(r-1)!(m+1-r)!} + \frac{m!}{r!(m-r)!} \\
&= \frac{rm! + (m+1-r)m!}{r!(m+1-r)!} \\
&= \frac{(r+(m+1-r))m!}{r!(m+1-r)!} \\
&= \frac{(m+1)(m)!}{r!(m+1-r)!} \\
&= \frac{n!}{(r)!(n-r)!} \\
&= \binom{n}{r}
\end{aligned}
$$

□

Hand-in Instructions

In the course directory /c/cs201/bin, there are five programs that support you in submitting your solution to the homework.

```
submit    assignment-number file(s)
unsubmit  assignment-number file(s)
check     assignment-number
protect   assignment-number file(s)
unprotect assignment-number file(s)
```

`submit` allows you to submit one of several files. For example

```
/c/cs201/bin/submit 1 hw1.sml hw1-examples.sml
```

submits your file `hw1.sml` and `hw1-examples.sml`. `check` can give you the peace of mind that your solution has really been submitted, and if you would like to make last minute changes to your solution, use `unsubmit` before submitting the updated version. `protect` and `unprotect` give you the power to protect/unprotect submitted solutions from deletion.