DM22 Exam Summer 2005

Sketch of possible solutions

1a: One straight-forward solution is the following:

```
selsort []     = []
selsort (x:xs) = m:(selsort rest)
  where
    m    = minimum (x:xs)
    rest = remove m (x:xs)

remove x []     = []
remove x (y:ys)
  | x == y    = ys
  | otherwise = y:(remove x ys)
```

Here, `minimum` is from the standard prelude. It can be replaced by `foldl1 min`. The function `remove` can also be found in the standard library `List` as `delete`. A variant is to find `m` and `rest` simultaneously:

```
selsort [] = []
selsort xs = m:(selsort rest)
  where
    (m,rest) = delmin xs

delmin [x]    = (x,[])
delmin (x:xs)
  | x <= y    = (x,xs)
  | otherwise = (y,x:ys)
  where
    (y,ys) = delmin xs
```

1b: A solution finding strings of a given length directly, using list comprehensions:

```
binstrN 0 = [""]
binstrN n = [ b:bs | b <- "01", bs <- binstrN (n-1) ]

binstrings = [ bs | len <- [0..], bs <- binstrN len
```

A solution finding strings of a given length from strings of previous length, using `iterate`:

```
addbits []     = []
addbits (x:xs) = (x++"0"):(x++"1"):addbits xs

binstrings = concat (iterate addbits [""])
```

A variant of this based on "process networks" (section 17.7 in Thompson) – note that only last line of definition of `addbits` is needed:

```
binstrings = "": addbits binstrings
```

2a: The predicates `append` and `prefix` are built-ins, but also appear in the textbook. The cut makes `frontRep` non-resatisfiable, which probably is the most natural - however, it is not necessary for our use of the predicate.

```
frontRep(L):-append(Pre,Rest,L),Pre\=[],prefix(Pre,Rest),!.
```

2b: Here is a version which builds solutions up from shorter solutions. Again, `member` is a built-in predicate, but also appears in the textbook. A simpler and slower version is to generate all strings over S and then check these for being solutions.

```
repFree([],0).
repFree([E|Y],N):-
  N>0,
  N1 is N-1,
  repFree(Y,N1),
  member(E,[1,2,3]),
  \+frontRep([E|Y]).
```

2c: The version below finds the number of solutions for each `N`, and then sum these up. Again, `findall` and `length` are built-in predicates, but also appear in the textbook (`length` under the name `listlen`).

```
count(N,R):- findall(R1,repFree(R1,N),L),length(L,R).

countLessThanEq(0,1).
countLessThanEq(N,R):-
  N>=1,
  count(N,R1),
  N2 is N-1,
  countLessThanEq(N2,R2),
  R is R1+R2.
```

Here is a version which uses that each solution of length at most `N` is generated exactly once when the version of `repFree` above is used to generate all solutions of length `N`. The code below keeps track of the count using a predicate `counter`. Note that `repFreeSave` essentially is a copy of `repFree`.

```
countLessThanEq(N,R):-
  asserta(counter(0)),
  findall(_,repFreeSave(_,N),_),
  counter(R),
  retract(counter(_)).

repFreeSave([],0):-incCounter.
repFreeSave([E|Y],N):-
  N>0,
  N1 is N-1,
  repFreeSave(Y,N1),
  member(E,[1,2,3]),
  \+frontRep([E|Y]),
  incCounter.

incCounter:-
  retract(counter(C)),
  C1 is C+1,
  asserta(counter(C1)).
```

3a: Two successful instantiations will be produced as results: `X = 1,Y = b` and `X = 1,Y = c`. The rest of the $2 \cdot 2 \cdot 3 = 12$ possible combinations of values of `v`, `u`, and `s` do not appear because of the cut, and because the two instances of `Y` must be the same value.

3b:

$$\forall X(\exists Y((a(X,Y) \vee b(Y)) \Rightarrow c(X)))$$

$$\forall X(\exists Y(\neg(a(X,Y) \vee b(Y)) \vee c(X)))$$

$$\forall X(\exists Y((\neg a(X,Y) \wedge \neg b(Y)) \vee c(X)))$$

$$\forall X((\neg a(X,f(X)) \wedge \neg b(f(X))) \vee c(X))$$

$$(\neg a(X,f(X)) \wedge \neg b(f(X))) \vee c(X)$$

$$(\neg a(X,f(X)) \vee c(X)) \wedge (\neg b(f(X)) \vee c(X))$$

Clausal form:

```
c(X):-a(X,f(X)).
c(X):-b(f(X)).
```

3c:

i) `X = t, Y = g(t), Z = t`

ii) `T = g(g(Z)), X = g(g(g(g(Z)))), Y = g(g(Z))`

iii) The two predicates do not unify: `X+Y` is a structure (not a number) having `X` as a subterm, and hence cannot unify with `X`.

3d:

i) `map zip :: [[a]] -> [[b] -> [(a,b)]]`

ii) `map . zip :: [a] -> [[b]] -> [[(a,b)]]`

4a: The proof is by induction on the length of `xs`. The base case is `xs = []`, which is easily proved. The induction step is proved by substitution of the definitions, followed by simple manipulations using the supplied equation in one of the steps.

4b: Two functions are equal if they have the same value at every argument (textbook, p. 193). Hence we must prove

```
(reverse . filter p) xs = reverse (filter p xs)
```

equal to

```
(filter p . reverse) xs = filter p (reverse xs)
```

for all lists `xs`. For finite lists, this has been done in part **a**. For fp-lists, we must prove an additional base case in the induction proof in part **a**, namely `xs = undef` (cf. textbook, p. 377). Since both functions use pattern matching on `xs`, they both return `undef` when `xs = undef`. Using this fact twice implies that both sides in the equation of **a** have the value `undef` and hence are equal. This extended induction proof proves the statement for fp-lists. Since we are dealing with an equation, this is enough to prove the statement for all infinite lists (textbook, p. 380).

4c: Assume `xs = [a,b,c,d,e]`, and that `p` happens to return `undef` on `c` but not on the remaining elements of `xs`. Then it is easy to argue that the left-hand side in part **a** is `reverse (p a):(p b):undef`, which again is `undef`, whereas the right-hand side is `filter p [e,d,c,b,a]`, which is `(p a):(p b):undef`, and hence different from the left-hand side.

So the equation in part **a** does not hold, and hence the equation in part **b** does not hold either.