```
module Final where

-- Imports everything from the FinalUltilities module.
import FinalUtilities
import Control.Applicative(liftA, liftA2, liftA3)

type State = Int

---for you to use in Section 4
plainWords =
["John","Mary","ate","bought","an","apple","books","yesterday","C","laughed","becau
se"]
whWords = ["who","what","why"]
qWords = ["Q"]
------------------------------------------------------------------
------------------------------------------------------------------
-- IMPORTANT: Please do not change anything above here.
--            Write all your code below this line.


-- Please submit your answer of Section 3 and Section 4.2 as a pdf.

----------Section 1
--1A
addToFront :: a -> SnocList a -> SnocList a
addToFront x sl = case sl of
    -- empty Snoclist, just return SnocList with x
    ESL -> (ESL:::x)
    -- add x to the front recursively and prepend the rest of Snoclist
    rest:::y -> (addToFront x rest) ::: y

--1B
toSnoc :: [a] -> SnocList a
toSnoc normal_list = case normal_list of
    -- normal_list is empty
    [] -> ESL
    -- normal_list is not empty, use addTofront as a helper function
    (x:rest) -> addToFront x (toSnoc rest)


--1C
forward :: (Eq a) => Automaton State a -> SnocList a -> State -> Bool
forward m w q =
    let (states, syms, i, f, delta) = m in
        case w of
            ESL -> elem q i
            (rest:::y) -> or (map (\q1 -> forward m rest q1 && elem (q1, y, q)
delta) states)

--1D
generatesViaForward :: (Eq a) => Automaton State a -> SnocList a -> Bool
generatesViaForward m w =
    let (states, syms, i, f, delta) = m in
    or (map (\q0 -> forward m w q0 && elem q0 f) states)


--1E
forwardGeneric :: (Semiring v) => GenericAutomaton st sy v -> SnocList sy -> st ->
v
```

```
forwardGeneric m w q =
    let (states, syms, i, f, delta) = m in
        case w of
            ESL -> i q
            (rest:::y) -> gen_or (map (\q1 -> forwardGeneric m rest q1 &&& delta
(q1, y, q)) states)

--1F
fViaForward :: (Semiring v) => GenericAutomaton st sy v -> SnocList sy -> v
fViaForward m w =
    let (states, syms, i, f, delta) = m in
        gen_or (map (\q1 -> forwardGeneric m w q1 &&& f q1) states)


----------Section 2
--2.1A

-- helper function
backwardSLG :: (Eq sy) => [(sy, sy)] -> [sy] -> sy -> Bool
backwardSLG m w q =
    case w of
    -- empty symbol list
    [] -> False
    -- one symbol
    [symbol] -> elem (symbol, q) m
    -- more than one symbol in list
    (y:rest) -> elem (y, head rest) m && backwardSLG m rest q


generatesSLG :: (Eq sy) => SLG sy -> [sy] -> Bool
generatesSLG m w =
    let (syms, i, f, bigrams) = m in
        case w of
            -- no way for an SLG to generate an empty String
            [] -> False
            -- one symbol
            [x] -> elem x i && elem x f
            -- more than one symbol in list
            (x:rest) -> elem x i && elem (last rest) f && backwardSLG bigrams rest
(last w)

--2.2A
slgStress::SLG SyllableTypes
slgStress = ([Stressed,Unstressed], [Stressed], [Stressed, Unstressed],
[(Stressed,Unstressed),(Unstressed,Unstressed)])

--2.2B
fsaHarmony :: Automaton Int SegmentPKIU
fsaHarmony = ([1,2], [P,K,I,U,MB], [1], [1,2], [(1, P, 1),
                                              (1, K, 1),
                                              (1, I, 1),
                                              (1, MB, 2),
                                              (2, P, 2),
                                              (2, U, 2),
                                              (2, MB, 1)])

--2.3A
slgToFSA :: SLG sy -> Automaton (ConstructedState sy) sy
slgToFSA (slg_alphabet, slg_i, slg_f, slg_sequences) =
```

```
    let
        make_transitions (firstSym, secondSym) = (StateForSymbol firstSym,
secondSym, StateForSymbol secondSym)
        -- transition from extra state to inital states
        extra_transitions = [(ExtraState, symbol, StateForSymbol symbol) | symbol
<- slg_i]
    in
        -- states
        (map StateForSymbol slg_alphabet ++ [ExtraState],
        -- alphabet
        slg_alphabet,
        -- intial state
        [ExtraState],
        -- Final State
        map StateForSymbol slg_f,
        -- transitions
        map make_transitions slg_sequences ++ extra_transitions)


--------------------- From Homework 5 for testing purposes ---------------------

allLists :: Int -> [a] -> [[a]]
allLists given_length given_list = case given_length of
    0 -> [[]]
    -- int greater than 0 (asssume non-negative)
    positiveNum -> [t : rest | t <- given_list, rest <- allLists (given_length - 1)
given_list]

under :: (Eq st, Eq sy) => TreeAutomaton st sy -> Tree sy -> st -> Bool
under (states, symbols, endingStates, transitions) (Node symbol daughters)
stateGoal =
    case daughters of
    -- check if transitions exists in list of transitons when no daughters in tree
    [] -> elem ([], symbol, stateGoal) transitions
    -- for all daughter states
    not_empty -> any (\daughter_states ->
            elem (daughter_states, symbol, stateGoal) transitions
            && and (zipWith (under (states, symbols, endingStates, transitions))
daughters daughter_states))
        (allLists (length daughters) states)

generatesFSTA :: (Eq st, Eq sy) => TreeAutomaton st sy -> Tree sy -> Bool
generatesFSTA (states, symbols, endingStates, transitions) tree =
    case endingStates of
        -- can't reach final states
        [] -> False
        -- check if ending state can be reached (check all ending states) using
under function
        (endState : rest) -> under (states, symbols, endingStates, transitions)
tree endState || generatesFSTA (states, symbols, rest, transitions) tree

--------------------------------------------------------------------------------


--------Section 4
--4.1A

data IslandState = None | BadAdjunct_encountered | Q_encountered | WH_encountered |
C_encountered | Both_encountered deriving (Eq, Show)
```

```
fsta_Island :: TreeAutomaton IslandState String
fsta_Island =
    -- states
    ([None, BadAdjunct_encountered, Q_encountered, WH_encountered, C_encountered,
Both_encountered],
    -- alphabet
    plainWords ++ whWords ++ qWords ++ ["*", "**"],
    -- ending states
    [Both_encountered],
    -- transitions
    [
    -- Nodes with "*"
    ([WH_encountered, WH_encountered], "*", WH_encountered),
     ([None, WH_encountered], "*", WH_encountered),
     ([WH_encountered, None], "*", WH_encountered),
     ([None, None], "*", None),
     ([Q_encountered, WH_encountered], "*", Both_encountered),
     ([Q_encountered, None], "*", None),
     ([None, Q_encountered], "*", None),
     ([Q_encountered, Q_encountered], "*", None),
     ([C_encountered, WH_encountered], "*", WH_encountered),
     ([C_encountered, Both_encountered], "*", Both_encountered),

     -- Nodes with "**", Adjuncts
     ([WH_encountered, WH_encountered], "**", WH_encountered),

     ([None, WH_encountered], "**", BadAdjunct_encountered),
     ([WH_encountered, None], "**", BadAdjunct_encountered),
     ([None, None], "**", None),
     ([Q_encountered, WH_encountered], "**", Both_encountered),
     ([Q_encountered, None], "**", None),
     ([None, Q_encountered], "**", None),

     -- "C" complementizer
     ([], "C", C_encountered),
     ([C_encountered, None], "*", Both_encountered)

    ] ++ map (\s -> ([], s, Q_encountered)) qWords
      ++ map (\s -> ([], s, WH_encountered)) whWords
      ++ map (\s -> ([], s, None)) plainWords
    )
```