Final Febrero 2025 (primer semana)

1. Recursion estructural.

```haskell
data Form = Prop String
            | And Form Form
            | Or Form Form
            | Neg Form
```

(A) Definir y dar el tipo de foldForm

```haskell
foldForm :: (String -> b) -- cProp
    -> (b -> b -> b)  -- cAnd
    -> (b -> b -> b) -- cOr
    -> (b -> b)     -- cNeg
    -> Form
    -> b
foldForm cProp cAnd cOr cNeg form = case form of
    Prop s -> cProp s
    And f1 f2 -> cAnd (rec f1) (rec f2)
    Or f1 f2 -> cOr (rec f1) (rec f2)
    Neg f -> cNeg (rec f)
    where rec = foldForm cProp cAnd cOr cNeg
```

(B) Definir fnn :: Form -> Bool -> Form usando foldForm.

Que pasa una formula x a forma normal negada si el booleano es True y pasa a la negacion de x a forma normal negada si el booleano es False.

Ej:

- `fnn (And (Prop "x") Neg(Or (Prop "y") Neg(Prop "z"))) True = (And (Prop "x") (And Neg(Prop "y") (Prop "z")))`
- `fnn (And (Prop "x") Neg(Or (Prop "y") Neg(Prop "z"))) False = (Or Neg(Prop "x") (Or (Prop "y") Neg(Prop "z")))`

```haskell
fnn :: Form -> Bool -> Form
fnn f b = foldForm
    (\s -> Prop s)                                          -- cProp
    (\r1 r2 -> if b then (And r1 r2) else (Or (Neg r1) (Neg r2)))   -- cAnd
    (\r1 f2 -> if b then (Or r1 r2) else (And (Neg r1) (Neg r2)))   -- cOr
    (\r -> if b then r else (Neg r))                        -- cNeg
    f
```

2. Demostración de igualdades por inducción estructural.

Dados

```haskell
alt :: (a -> b) -> (a -> b) -> [a] -> [b]
alt f g [] = []                     --{A0}
alt f g (x:xs) = f x : alt g f xs  --{A1}
```

Demostrar `alt g1 g2 . alt f1 f2 = alt (g1 . f1) (g2 . f2)`.

Queremos ver que vale $\forall \text{l} :: [\text{a}] P(l)$:

$P(l) \equiv (\text{alt g1 g2}) \text{ . } (\text{alt f1 f2}) \text{ l} = \text{alt } (\text{g1 . f1}) \text{ } (\text{g2 . f2}) \text{ l}$

Para esto vemos que vale para todos los constructores de listas, asumiendo como HI que vale para las sublistas de l.

Lista vacía

$P([]) \equiv (\text{alt g1 g2}) \text{ . } (\text{alt f1 f2}) \text{ } [] = \text{alt } (\text{g1 . f1}) \text{ } (\text{g2 . f2}) \text{ } []$

```haskell
-- IZQ
= (alt g1 g2) . (alt f1 f2) []
= (alt g1 g2 (alt f1 f2 [])) -- composicion
= (alt g1 g2 []) -- A0
= []

-- DER
= alt (g1 . f1) (g2 . f2) []
= []
```

Luego ambos lados reducen a la misma forma normal, entonces vale P([]).

Lista con elementos

$P((x : xs)) \equiv (\text{alt g1 g2}) \text{ . } (\text{alt f1 f2}) \text{ } (\text{x:xs}) = \text{alt } (\text{g1 . f1}) \text{ } (\text{g2 . f2}) \text{ } (\text{x:xs})$

```haskell
-- IZQ
= (alt g1 g2) . (alt f1 f2) (x:xs)
= (alt g1 g2 (alt f1 f2 (x:xs))) -- composicion
= (alt g1 g2 (f1 x : alt f2 f1 xs)) -- a1
= g1 (f1 x) : alt g2 g1 (alt f2 f1 xs) -- a1

-- DER
= alt (g1 . f1) (g2 . f2) (x:xs)
= (g1 . f1) x : alt (g2 . f2) (g1 .f1) xs

-- De acá vemos que:
-- |-> g1 (f1 x) == (g1. f1) x -- Vale por definicion de composicion
-- |-> alt g2 g1 (alt f2 f1 xs) == alt (g2 . f2) (g1 .f1) xs -- Vale por ser la HI (hipotesis inductiva)

-- Entonces IZQ = DER
```

Luego, QED $\forall \text{l} :: [\text{a}] P(l)$.

3. Deducción natural.

> En cuaderno

4. Programación lógica.

En prolog definir λ-terminos como:

- var(X) donde X es un numero natural. Representa al uso de la variable numero X.
- lam(X, M) donde X es un numero natural y M es un λ-termino. Representa la ligadura de la variable numero X con respecto al λ-termino M.
- app(M, N) donde M y N son λ-terminos. Representa a la aplicacion.

Ej:

```
lam(1, lam(2, lam(3, app(var(1), app(var(2), var(3)))))
-- esto es equivalente al termino λf. λg. λx. f (g x)
```

(A) Definir variablesLibres(+M, -L) que instancia en una lista L las variables libres del termino M.

- Ej: variablesLibres(lam(1, app(var(1), var(2))), L) instancia L = [2].

```
% La defino por inducción en la estructura de términos lambda:

variablesLibres(var(X), [X]).
variablesLibres(app(M, N), L) :- variablesLibres(M, V1), variablesLibes(N, V2), L is V1++V2.
variablesLibres(lam(X, M), L) :- variablesLibres(M, V1), sacar(X, V1, L).

sacar(X, [], []).
sacar(X, [X|XS], XS).
sacar(X, [Y|Ys], [Y|Ls]) :- sacar(X, Ys, Ls).
```

(B) Definir tamano(+M, -T) que calcula el tamano de un termino. var(X) suma 1. lam(X, M) suma 1 + tamano(M). app(M, N) suma 1 + tamano(M) + tamano(N).

- Ej: tamano(app(var(1), lam(1, var(2)))) instancia T = 4.

```
tamaño(var(X), 1).
tamaño(lam(M, N), Z) :- tamaño(N, R), Z is R+1.
tamaño(app(M, N), Z) :- tamaño(N, R), tamaño(M, Q), Z is Q+R+1.
```

(C) Definir generarLambdaTerminos(+xs, -M) que dada una lista de numeros naturales XS instancia en M λ-terminos infinitos. Sugerencia: Crear los λ-terminos en base al tamano.

```
generarLambdaTerminos(XS, M) :-
    desde(0,S), % Genero el tamaño del lambda termino
    generarLambdaTerminoTamaño(S, M, XS). % Genero un lambda término de tamaño S con elementos de XS

generarLambdaTerminoTamaño(1, var(X), XS) :- member(X, XS).
generarLambdaTerminoTamaño(S, app(M, N), XS) :-
    S > 1,
    Q is S-1,
    numerosQueSuman(Q, A, B),
    generarLambdaTerminoTamaño(A, M, XS),
    generarLambdaTerminoTamaño(B, N, XS).
generarLambdaTerminoTamaño(S, lam(X, N), XS) :-
    S > 1,
    Q is S-1,
    member(X,XS),
    generarLambdaTerminoTamaño(XS, N, Q).

numerosQueSuman(N, A, B) :- between(0, N, A), B is N-A.
```