# Instituto Tecnológico de Costa Rica

Leguajes de Programación - Proyecto 1
Procesador para fomulas logicas con variables
I Semestre - 2020

Jasson Gonzalez Torrez

201116046

David Jose Espinoza Soto

2016012024

Anthony Andres Ulloa Martinez

2018290801

Julio 2020

# 1. vars

## 1.1. Estrategia

**- La función vars es responsable de extraer las variables sin duplicados de una proposición. Para lograr esto se recorre el árbol de expresión hasta llegar a sus hojas que representan las varibles. Como caso base, si se encuentra una variable esta se retorna.**

**- Una vez recorrido todo el árbol de expresión y obtenidas todas las variables presentes en el árbol de expresión, se procede a eliminar los elementos duplicados**

**-David Espinoza: Esta fué la primera función que escribí en Moscow ML, y por lo tanto la base estratégica de todas las demas funciones escritas por mi persona. Al principio, no entendiamos muy bien la estructura base de 'datatype Proposicion', al final nos comprendimos que su estructura se asemejaba a los arboles binarios que estudiamos en POO.**

**- Una vez que empezamos a experimentar creando Proposiciones, para estudiar su estructura, decidi imitar la función evalProp del evaluador de propociciones simples, pero modificándola para que devolviera las variables que encontrara.**

**- Una vez teniendo una lista de las variables, solo faltaba asegurarse que cada variable fuera única. Por eso busqué funciones para filtar listas, y simplemente las aplicamos al resultado.**

# 2. eval_prop

## 2.1. Estrategia

**- Al trabajar en la función evalprop se tomaron en cuenta dos posibles soluciones:**

1. Tomar la proposición original e iterarla, generando paralelamente una expresión que reemplaza las variables por sus respectivos valores booleanos

2. Modificar la función eval_prop para constantes, añadiendo un caso base adicional al recibir variables y después retornar la conversión de la variable a su respectivo valor booleano

**- Se tomó como opción más viable la segunda opción. Agregando funciones para obtener el último elemento de una lista y una función para obtener un elemento de una lista de tuplas basado en una llave.**

# 3. gen_bools

## 3.1. Estrategia

**- La funcion genbools es responsable de generar arreglos que contienen todas las posibles asignaciones de valores booleanos para las variables que se encuentran en una proposición.**

**- Al trabajar en la función genbools se tomaron en cuenta dos posibles soluciones:**

1. Mediante llamadas recursivas de 0 a 2**n, donde n representa el número de variables proposicionales, generar el equivalente en binarion y convertir el resultado en un arreglo de elementos booleanos

2. La otra aproximación a la solución fue generar la combinación de valores booleanos para 1 proposicional y para 2 variables proposicionales. Después se pueden agregar de forma recursiva las combinaciones para 1 elemento y unirlas con el arreglo booleano de n-1

**- La opción que usa recursión nació como fruto de varias horas de análisis de casos que se realizaron en papel. Se buscaba un patrón que se pudiera generalizar para el caso base, y de ahí construir los casos más complejos. Cuando se determinó que la función con n=1 era un caso especial, se tomó n=2 como la base de la solución, y determinamos que para cualquier otro n se construiria el arreglo añadiendo los valores booleanos a los arreglos optenido de n-1.**

# 4. as_vals

## 4.1. Estrategia

**- La función as_vals es responsable de generar asignaciones de variables proposicionales con las posibles combinaciones de valores booleanos. El resultado de la funcion es una lista de listas de string * bool.**

**- Se hicieron varios intentos utlizando como base la funcion zipP que el profesor brindó para hacer la asignacion de valores, se encontraron varios problemas debido a que no se podía mantener el largo de la primera lista; la de variables. Además también se encontraron problemas con el hecho de que se estaba trabajando con una lista de listas; la de lista de booleanos, por lo que habia que tratarla de forma diferente.**

**- La solución para los problemas antes mencionados se dio dividiendo la lista de booleanos y después enviando cada una de las sublistas con la lista original de variables sin el primer valor a una función auxiliar que se encargaba de ir asignando. En la siguiente iteración se hizo uso de x:xs para mantener la lista de variables intacta y continuar con el resto de la lista de booleanos.**

**- La idea se dio al ver el comportamiento de la funcion zipP del profesor, se hicieron varios intentos para ver como se desarrollaba con listas de listas hasta que se dio con la solución final.**

# 5. taut

## 5.1. Estrategia

**- La función taut es responsable de evaluar la Proposición y determinar si es una Tautología, o no. Y en caso de que no sea una tautología, se debe especificar cuál o cuales son los valores que la falsifican.**

**- Esta función recibe una proposición y utiliza las funciones anteriormente mencionadas para evaluarla. Solo por motivos de estética la función taut no evalua la proposición tal cual, sino que prepara el ambiente y se lo delega a una función auxiliar, que recursivamente evalua la proposición.**

**- Esta función auxiliar recibe como parámetros a la proposición y un arreglo de valores que representan cada uno de las posibles combinaciones de valores que pueden adoptar las variables de la proposición. La función asume que siempre recibirá un arreglo de opciones, asi que asume que si el arreglo llega vacío significa que no encontró falacias en la proposición.**

**- Por esto el caso base es una lista vacía, pero si no lo es entonces evalua la propoción con el primer elemento de las combinaciones. En caso de que no sea una tautología, inmediatamente devuelve un string con los valores que la falsifican, y en caso contrario se llama recursivamente usando el resto de la lista de combinaciones.**

# 6. simpl

## 6.1. Estrategia

**- La función simpl simplifica una proposición lógica mediante el uso de una de serie de reglas.**

**- Para generar una proposición simplificada, se crearon diferentes funciones que aplican reglas de lógica específicas. La función simpl se encarga de ejecutar estas sub-funciones de forma secuencial y retornar el resultado final.**

**- Cada una de las sub_funciones empleadas en simp lleva a cabo las reglas de simplificacion y se hacen en un orden especifico para simplificar de la mejor manera la proposicion.**

# 7. Pruebas Realizadas

**- Proceda a revisar el archivo Pruebas.sml donde se encontraran con cada una de las pruebas para sus funciones correspondientes.**

# 8. Instrucciones para ejecutar el programa

**- Ir a la carpeta Codigo fuente.**

**- Ejecute el archivo proyecto.sml**

**- Defina una proposición y ejecute la función taut**

**- Defina una proposición y ejecute la función simpl**

# 9. Estado del programa

**- Fue posible implementar las funciones solicitadas de forma existosa. Gracias a a las oportunas consultas realizadas al profesor, se pudo dar un desarrollo eficiente de los diferentes componentes del programa.**

**- Uno de los problemas que el equipo tuvo que superar fue el poco conocimiento en SML y los lenguajes funcionales. Por medio de la investigación y el uso de los manuales de SML fue posible encontrar las formas de llevar a cabo las funciones requeridas.**

# 10. Análisis de los resultados obtenidos

**- La asignacion fue completada con éxito, ya que se pudo realizar cada uno de las funciones correspondientes. La funcion simpl se llevo un poco mas de tiempo, esto debido a la cantidad de subfunciones que tuvieron que crearse para que funcionara correctamente. El cambio a evalprop fue relativamente sencillo, las funciones vars, gen_bools y as_vals iban muy de la mano pero se completaron sin problemas. En el caso de taut se llevo algo mas de tiempo debido a las implicaciones detras de la misma. En general la carga de cada una de las funciones no fue muy pesada y se logro completar satisfactoriamente cada una de ellas.**

## 10.1. Conclusiones

**- El problema planteado permitió comprender las características de SML y de lenguajes funcionales. Se refrescaron conceptos de lógica booleana así como prácticas de programación que son requeridas en diferentes momentos de la carrera profesional. La asignacion fue útil para pensar en como solucionar problemas algunas veces complejos utilizando la fragmentacion del mismo mediante recursion. El proyecto estimuló la desarrollo de habilidades blandas y permitió mejorar las capacidades de trabajo en equipo.**

# 11. Experiencia de trabajar Standard ML y el paradigma de programación funcional

## 11.1. Retos

**- Aunque existe mucha documentación sobre el lenguaje, fue complicado encontrar fuentes que aportaran ayuda para la solución. En parte se debe a que varios de los ejemplos estaban enfocados a las capacidades más fuertes del lenguaje dejando de lado las funcionalidades más simples que permitieran aproximarse al problema dado, por lo tanto fue necesario utilzar varias estrategias para poder formular una solucion viable.**

## 11.2. Ventajas

**- La inferencia de tipos fue de gran ayuda y nos permitió realizar todas las combinaciones posibles para aplicar los cambios necesarios a las proposiciones. Fue bastante útil para comprender de una mejor manera el uso de la recursion, en lenguajes parecido a SML.**

# 12. Tareas realizadas por cada miembro del grupo de trabajo.

## 12.1. David Espinoza

**- Participacíon activa en las reuniones, que se realizaron frecuentemente para coordinar y aclarar el desarrollo de diversas funciones. Creé el repositorio de GitHub en donde se desarrollo el proyecto. Desarrollé la función vars, taut, gen_bool, y coolabora en el desarrollo de simpl al realizar doble_negacion, neutro e idempotencia.**

## 12.2. Jasson Gonzalez

**- Participación activa en la discusión para la implementación de las diferentes funcionalidades a lo largo del desarrollo del proyecto. Propuestas para las funciones iniciales a ser utilizadas en taut. Trabajo en el desarrollo de eval_prop y sus funciones asociadas. Trabajo en la definición de patrones especificos para la ley de morgan.**

## 12.3. Anthony Ulloa

**- Participante en las reuniones en las que se discutían las ideas principales del proyecto, asi como también la creación de la funcion as_vals y cuatro de las siete funciones requeridas para la utilización de la función simpl, siendo estas com, dis, aso y parte de la de_morga, esta última se completó con la ayuda del grupo.**

# 13. Código fuente

## 13.1. Datatype proposicion

```
datatype Proposicion =
   constante   of bool
|  variable    of string
|  negacion    of Proposicion
|  conjuncion  of Proposicion * Proposicion
|  disyuncion  of Proposicion * Proposicion
|  implicacion of Proposicion * Proposicion
| equivalencia of Proposicion * Proposicion
;

nonfix ~:
val ~: = negacion

infix 7 :&&:
val (op :&&:) = conjuncion

infix 6 :||:
val (op :||:) = disyuncion

infix 5 :=>:
val (op :=>:) = implicacion

infix 4 :<=>:
val (op :<=>:) = equivalencia

;
(* El ejemplo de entrada *)
val p = variable "p";
val q = variable "q";
val f = constante false;
val t = constante true;
val prop1 = p :=>: q :<=>: ~: p :||: q;
val prop2 = f :=>: p :<=>: q :=>: ~: f;
```

## 13.2. vars

```
fun filter p []      = []
|   filter p (x::xs) = if p x then x :: filter p xs else filter p xs
;


(* nub obtiene una lista sin duplicados a partir de una lista arbitraria *)
fun nub []      = []
|   nub (x::xs) = x :: (nub (filter (fn y => x <> y) xs))
;
```

```
(* vars *)
type str = string;

fun aux_vars prop =
case prop of
constante valor
=> [ ]
| variable valor
=> [valor]
| negacion prop1
=> aux_vars prop1
| conjuncion (prop1, prop2)
=> let val valor1 = aux_vars prop1
   and valor2 = aux_vars prop2
   in valor1 @ valor2
   end
|   disyuncion (prop1, prop2)
=> let val valor1 = aux_vars prop1
   and valor2 = aux_vars prop2
   in valor1 @ valor2
   end
| implicacion (prop1, prop2)
=> let val valor1 = aux_vars prop1
   and valor2 = aux_vars prop2
   in valor1 @ valor2
   end
| equivalencia (prop1, prop2)
=> let val valor1 = aux_vars prop1
   and valor2 = aux_vars prop2
   in valor1 @ valor2
   end
;
```

### 13.3. gen_bools

```
fun add [x] xs = [x] @ xs;

fun map f [] = []
| map f (x::xs) = f(x) :: map f xs;

(* - map (add [true] ) [[true, true], [false, false]];                    *)
(* > val it = [[true, true, true], [true, false, false]] : bool list list *)

fun gen_bools 1 = [[true, false]]
|   gen_bools 2 = [[true, true], [true, false], [false, true], [false, false]]
|   gen_bools n = (map (add [true] ) (gen_bools (n-1))) @ (map (add [false] ) (gen_bools (
```

### 13.4. as_val

```
fun add []    []    = []
|   add (x)   (y)   = (x) @ (y)
;

fun as_val_aux []        []              = []
|   as_val_aux (x :: xs) (y :: ys)       = (x, y) :: as_val_aux xs ys
|   as_val_aux []        _               = []
|   as_val_aux _         _               = []
;

fun as_val   []          []              = []
|   as_val   (x :: xs)   ((y::js) :: ys)  = add [(x, y)] (as_val_aux xs js) :: as_val (x :
|   as_val   []          _               = []
|   as_val   _           _               = []
;
```

### 13.5. eval_prop

```
fun last(xs) =
case xs of
    [] => raise List.Empty
    | (x::[]) => x
    | (_::xs') => last(xs')

fun get_value (nombreVariable: string, lista: (string*bool) list) =
    if null lista
    then false
    else if #1 (hd lista) = nombreVariable
    then #2 (hd lista)
    else get_value (nombreVariable, tl lista);

get_value("r", [("p", false), ("q",true), ("r",false)]);
get_value("q", [("p", false), ("q",true), ("r",true)]);

fun evalProp (prop, contexto) =
  case prop of
   constante valor
        => valor
  |
    variable valor
        => get_value(valor, contexto)
  | negacion prop1
       => not (evalProp (prop1, contexto))
  | conjuncion (prop1, prop2)
       => let val valor1 = evalProp (prop1, contexto)
              and valor2 = evalProp (prop2, contexto)
          in  valor1 andalso valor2
          end
```

```
  | disyuncion (prop1, prop2)
      => let val valor1 = evalProp (prop1, contexto)
             and valor2 = evalProp (prop2, contexto)
         in  valor1 orelse valor2
         end
  | implicacion (prop1, prop2)
      => let val valor1 = evalProp (prop1, contexto)
             and valor2 = evalProp (prop2, contexto)
         in  case (valor1, valor2) of
               (true, false) => false
             | _             => true
         end
  | equivalencia (prop1, prop2)
      => let val valor1 = evalProp (prop1, contexto)
             and valor2 = evalProp (prop2, contexto)
         in  valor1 = valor2
         end
;
```

### 13.6. taut

```
fun print_tuple (x,y) =
if y = true
then x ^ " = true"
else x ^ " = false";

fun print_context [] = " "
| print_context (x::xs) = print_tuple x ^ ", " ^ print_context xs;

(* necesito una funcion para saber cuantas variables ahi *)
fun cont_vars [] = 0
| cont_vars (x::xs) = 1 + cont_vars xs;

fun aux_taut_logic prop [] = "Es una Tautologia"
| aux_taut_logic prop (x::xs) =
if (evalProp (prop, x) )
then aux_taut_logic prop xs
else "No es Tautologia por que; " ^ print_context x ^ " hacen falsa la propocicion."

(* fun aux_taut proporciona un ambiente *)

fun aux_taut prop =
if (cont_vars (vars prop)) = 0
then aux_taut_logic prop [[("exception",true)]]
else aux_taut_logic prop (as_val (vars prop) (gen_bools(cont_vars (vars prop))) );

(* funcion taut *)
fun taut prop = aux_taut prop;
```

## 13.7. asociativa

```
fun aso prop =
case prop of
constante valor
=> constante valor
| variable valor
=> variable valor
| negacion prop1
=> negacion (aso prop1)
    | conjuncion (conjuncion (prop1, prop2), prop3)
=> conjuncion (aso prop1, conjuncion (aso prop2, aso prop3))
| conjuncion (prop1, conjuncion (prop2, prop3))
=> conjuncion (conjuncion (aso prop1, aso prop2), aso prop3)
    |   disyuncion (disyuncion (prop1, prop2), prop3)
            => disyuncion (aso prop1, disyuncion (aso prop2, aso prop3))
|   disyuncion (prop1, disyuncion (prop2, prop3))
=> disyuncion (disyuncion (aso prop1, aso prop2), aso prop3)
| implicacion (prop1, prop2)
=> implicacion (aso prop1, aso prop2)
| equivalencia (prop1, prop2)
=> equivalencia (aso prop1, aso prop2)
| _
=> prop
;
```

## 13.8. cont_elem_prop

```
(* funcion contar elementos de de una proposicion *)

fun cont_elem_prop prop =
case prop of
constante valor
=> 1
| variable valor
=> 1
| conjuncion (prop1, prop2)
=> 1 + cont_elem_prop prop1 + cont_elem_prop prop2
|   disyuncion (prop1, prop2)
=> 1 + cont_elem_prop prop1 + cont_elem_prop prop2
| negacion prop1
=> 1 + cont_elem_prop prop1
| implicacion (prop1, prop2)
=> 1 + cont_elem_prop prop1 + cont_elem_prop prop2
| equivalencia (prop1, prop2)
=> 1 + cont_elem_prop prop1 + cont_elem_prop prop2
;
```

## 13.9. Conmutativa

```
fun com prop =
case prop of
constante valor
=> constante valor
| variable valor
=> variable valor
| negacion prop1
=> negacion (com prop1)
| conjuncion (prop1, prop2)
=> conjuncion (com prop2, com prop1)
|   disyuncion (prop1, prop2)
=> disyuncion (com prop2, com prop1)
| implicacion (prop1, prop2)
=> implicacion (com prop1, com prop2)
| equivalencia (prop1, prop2)
=> equivalencia (com prop1, com prop2)
;
```

## 13.10. de_morgan

```
fun de_morgan prop =
case prop of
constante valor
=> constante valor
| variable valor
=> variable valor
        | negacion prop1
            => ( case prop1 of
               conjuncion(prop1, prop2) => disyuncion(negacion(de_morgan(prop1)), negacion
               | disyuncion(prop1, prop2) => conjuncion(negacion(de_morgan(prop1)), negaci
   | _ => negacion (de_morgan prop1)
   )
| conjuncion (negacion prop1, negacion prop2)
=> negacion (disyuncion (de_morgan prop1, de_morgan prop2))
|   disyuncion (negacion prop1, negacion prop2)
=> negacion (conjuncion (de_morgan prop1, de_morgan prop2))
| implicacion (prop1, prop2)
=> implicacion (de_morgan prop1, de_morgan prop2)
| equivalencia (prop1, prop2)
=> equivalencia (de_morgan prop1, de_morgan prop2)
| conjuncion(prop1, prop2)
=> negacion(disyuncion(negacion(de_morgan (prop1)), negacion(de_morgan (prop2))))
| disyuncion(prop1, prop2)
=> negacion(conjuncion(negacion(de_morgan (prop1)), negacion(de_morgan (prop2))))
;
```

### 13.11. Distributiva

```
fun dis prop =
case prop of
constante valor
=> constante valor
| variable valor
=> variable valor
| negacion prop1
=> negacion (dis prop1)
|   conjuncion (disyuncion (prop1, prop2), disyuncion (prop3, prop4))
=> if prop1 = prop3 then
    disyuncion (dis prop1, conjuncion(dis prop2, dis prop4))
   else
    conjuncion (disyuncion (dis prop1, dis prop2), disyuncion (dis prop3, dis prop4))
    | disyuncion (conjuncion (prop1, prop2), conjuncion (prop3, prop4))
=> if prop1 = prop3 then
conjuncion (dis prop1, disyuncion(dis prop2, dis prop4))
   else
    disyuncion (conjuncion (dis prop1, dis prop2), conjuncion (dis prop3, dis prop4))
| implicacion (prop1, prop2)
=> implicacion (dis prop1, dis prop2)
| equivalencia (prop1, prop2)
=> equivalencia (dis prop1, dis prop2)
| _
=> prop
;
```

## 13.12. Doble negacion

```
fun doble_neg prop =
case prop of
constante valor
=> constante valor
| variable valor
=> variable valor
| negacion( negacion( prop1))
=> doble_neg prop1
| negacion prop1
=> negacion (doble_neg prop1)
| conjuncion (prop1, prop2)
=> conjuncion (doble_neg prop1, doble_neg prop2)
|   disyuncion (prop1, prop2)
=> disyuncion (doble_neg prop1, doble_neg prop2)
| implicacion (prop1, prop2)
=> implicacion (doble_neg prop1, doble_neg prop2)
| equivalencia (prop1, prop2)
=> equivalencia (doble_neg prop1, doble_neg prop2)
;
```

## 13.13. idempotencia

```
fun idempotencia prop =
case prop of
constante valor
=> constante valor
| variable valor
=> variable valor
| conjuncion ( prop1 , prop2 ) =>
if (prop1 = prop2)
then idempotencia prop1
else conjuncion (idempotencia prop1, idempotencia prop2)
| disyuncion ( prop1 , prop2 ) =>
if (prop1 = prop2)
then idempotencia prop1
else disyuncion (idempotencia prop1, idempotencia prop2)
| negacion prop1
=> negacion (idempotencia prop1)
| implicacion (prop1, prop2)
=> implicacion (idempotencia prop1, idempotencia prop2)
| equivalencia (prop1, prop2)
=> equivalencia (idempotencia prop1, idempotencia prop2)
;
```

### 13.14. neutro

```
fun isVariable prop =
case prop of
variable _  => true
|          _  => false
;

fun neutro prop =
case prop of
constante valor
=> constante valor
| variable valor
=> variable valor
| conjuncion (prop1, constante true)
=> neutro prop1
| conjuncion (constante true, prop2)
=> neutro prop2
| conjuncion (prop1, prop2)
=> conjuncion (neutro prop1, neutro prop2)
|   disyuncion (prop1, constante false)
=> neutro prop1
|   disyuncion (constante false, prop2)
=> neutro prop2
| disyuncion (prop1, prop2)
=> disyuncion (neutro prop1, neutro prop2)
| negacion prop1
=> negacion (neutro prop1)
| implicacion (constante true, prop2)
=> neutro prop2
| implicacion (prop1, prop2)
=> implicacion (neutro prop1, neutro prop2)
| equivalencia (prop1, prop2)
=> equivalencia (neutro prop1, neutro prop2)
;
```

### 13.15. simpl

```
fun simp prop =
    aso(com(de_morgan(dis(doble_neg(idempotencia(neutro(prop)))))))
;
```

# Referencias


[1] L. C Paulson. *ML for the working programmer*. Cambridge University Press, 1996.

[2] Hansen Rischel. *Ïntroduction to Programming using SML". Addison-Wesley.* Web, 1999.

[3] Robert Harper. Programming in standard ml.

[4] James Baker Leo Zovic Chris Wilson Simon Shine, David Pedersen. Learn x in y minutes.

[5] Keunwoo Lee. Cse 341 : Programming languages winter 2004 - university of washington.

[6] Edwin Dalorzo. Learning sml - basic list functions.

[7] James Baker Leo Zovic Chris Wilson Simon Shine, David Pedersen. Learn standard ml in y minutes.