v.7.6

Trabajo Práctico 6

Programación I

En este trabajo práctico se propone resolver una consigna integradora, con el objetivo de aplicar los conocimientos adquiridos a lo largo del curso.

1 Enunciado

En la lógica clásica, una *proposición* es un enunciado al que, bajo ciertas reglas preestablecidas, es posible atribuirle un valor de verdad: verdadero o falso. Por ejemplo, "*LCC es una carrera de tres años*", "*Las naranjas son frutas*" y "*Santa Fe tiene 19 departamentos*" son ejemplos de proposiciones. A partir de proposiciones simples o atómicas se pueden construir proposiciones más complejas utilizando los conectivos lógicos. Por ejemplo, "*Si Santa Fe tiene 19 departamentos, entonces las naranjas son frutas*".

Los operadores lógicos más usados son la conjunción ($\wedge$), la disyunción ($\vee$), la negación ($\neg$), la implicancia ($\rightarrow$) y la equivalencia ($\leftrightarrow$).

Cuando trabajamos en lógica clásica, el significado de estos operadorados queda totalmente determinado a partir de su *tabla de verdad*. Una tabla de verdad nos dice, para cada posible valor de verdad de las proposiciones más simples, cuál es el valor de verdad de la proposición. Ya vimos algunas tablas de verdad en la Práctica 0, tanto para la conjunción (`and`), disyunción (`or`) y negación (`not`). Las siguientes tablas de verdad se corresponden con la implicancia y la equivalencia:

p	**q**	**(p $\rightarrow$ q)**
`#true`	`#true`	`#true`
`#true`	`#false`	`#false`
`#false`	`#true`	`#true`
`#false`	`#false`	`#true`

p	**q**	**(p $\leftrightarrow$ q)**
`#true`	`#true`	`#true`
`#true`	`#false`	`#false`
`#false`	`#true`	`#false`
`#false`	`#false`	`#true`

Ejercicio 1. Para comenzar, diseñe las funciones `implica` y `equivalente`, que dados dos valores booleanos se comporten como la tabla de verdad de los operadores $\rightarrow$ y $\leftrightarrow$, respectivamente.

Cada fila de la tabla de verdad representa una valuación. Esto es, una posible asignación de valores de verdad para las variables involucradas en la fórmula. Observemos que si una fórmula tiene *n* variables, entonces la tabla de verdad tendrá 2^n filas.

Decimos que una fórmula es:

- una *tautología*, si es verdadera para todas las posibles valuaciones;
- una *contradicción*, si es falsa para todas las posibles valuaciones;

- *satisfactible*, si es verdadera para al menos una valuación;

Consideremos las siguientes fórmulas proposicionales:

$$A = ((p_1 \to p_3) \land (p_2 \to p_3)) \leftrightarrow ((p_1 \lor p_2) \to p_3)$$
$$B = ((p_1 \land p_2) \to p_3) \leftrightarrow ((p_1 \to p_3) \land (p_2 \to p_3))$$
$$C = (\neg p_1 \lor \neg p_2) \leftrightarrow (p_1 \land p_2)$$

No es difícil observar que estas fórmulas son: una tautología, una fórmula satisfactible y una contradicción, respectivamente. Sin embargo, para verificarlo, sería bueno que en este punto se detenga y haga con lápiz y papel la tabla de verdad de cada una de ellas.

Estamos interesados en diseñar los predicados `tautología?`, `contradicción?` y `satisfactible?`, que nos permitan clasificar fórmulas proposicionales. Haremos esto evaluando una fórmula en todas las posibles valuaciones. Es decir, el mismo método que siguen las tablas de verdad.

Para esto, deberíamos ser capaces de:

- Poder representar valuaciones
- Poder representar fórmulas proposicionales
- Poder evaluar una fórmula en una valuación

1.1 Representando valuaciones

Para representar una valuación, podemos usar listas de booleanos. Por ejemplo, para la fórmula A, la lista `(list #t #f #t)` es una valuación posible, donde el primer elemento (`#t`) corresponde a la variable p_1, el segundo elemento (`#f`) corresponde a p_2 y el tercero (`#t`) a p_3. Si evaluamos la proposición en esta lista, obtenemos `#t`.

La lista `(list #t #t)` es una valuación posible para la fórmula C. Para esta valuación obtendremos `#f` como resultado.

Cada valuación representa una fila en la tabla de verdad de la proposición.

Ejercicio 2. Diseñe una función `valuaciones` que, dado un número natural `n`, genere todas las posibles valuaciones que existen con `n` variables proposicionales. Por ejemplo, para `n=3` debería devolver:

```
(list
 (list #false #false #false)
 (list #false #false #true)
 (list #false #true #false)
 (list #false #true #true)
 (list #true #false #false)
 (list #true #false #true)
 (list #true #true #false)
 (list #true #true #true))
```

o bien una permutación de esa lista de tamaño de `8`.

1.2 Representando y evaluando fórmulas

La fórmula proposicional *A* puede pensarse como una función que, dados tres valores booleanos `a`, `b`, `c`, devuelve el valor booleano que resulta de sustituir p_1 por `a`, p_2 por `b`, y p_3 por `c`.

Es decir, *A* podría representarse mediante una función de tres argumentos booleanos. Sin embargo, *C* debería representarse como una función de dos argumentos, puesto que sólo involucra dos variables proposicionales. Para hacer más uniforme la representación, utilizaremos listas de booleanos como argumentos de las fórmulas, y de esta manera podremos representar funciones proposicionales con cualquier cantidad de variables.

Usaremos la siguiente signatura para fórmulas:

```
List (Boolean) -> Boolean.
```

Con esta idea, podemos representar la fórmula *A* como sigue:

```
; A : List(Boolean) -> Boolean
(define
  (A l)
  (equivalente (and (implica (first l) (third l))
                    (implica (second l) (third l)))
               (implica (or (first l) (second l))
                        (third l)))))
```

Puede parecer complicado, pero es el precio que tenemos que pagar para tener una signatura más uniforme y una evaluación más simple. También nos podríamos ayudar usando `let`:

```
; A : List(Boolean) -> Boolean
(define
  (A l)
  (let ([p1 (first l)]
        [p2 (second l)]
        [p3 (third l)])
  (equivalente (and (implica p1 p3)
                    (implica p2 p3))
               (implica (or p1 p2)
                        p3))))
```

Las expresiones `let` nos permiten asociar uno o más nombres (por ejemplo, `p1`) a una o más expresiones (por ejemplo, `(first l)`). Para el ejemplo, los nombres `p1`, `p2` y `p3` sólo pueden usarse dentro de la definición de `A`.

Con esta definición, podemos evaluar:

```
(A (list #t #f #t))
> #t
```

Ejercicio 3. Defina las funciones `B` y `C` para las otras dos fórmulas que aparecen en los ejemplos. Obtenga el valor de verdad para cada una de ellas en un ejemplo concreto de valuación.

Ejercicio 4. Utilizando los ejercicios anteriores, diseñe una función `evaluar` que, dada una fórmula proposicional `P: List (Boolean) -> Boolean` y la cantidad de variables `n` que utiliza `P`, devuelva una lista con 2^n booleanos, que son el resultado de evaluar `P` en cada una de las posibles valuaciones. Es decir, que devuelva una lista con la última columna de la tabla de verdad de `P`.

Ejercicio 5. Diseñe los predicados `tautología?`, `contradicción?` y `satisfactible?`.

Ejercicio 6. El archivo tests.txt contiene algunos casos de tests que la/o ayudarán a detectar posibles errores. Asegúrese que su código pase dichos casos de tests. Para ello, luego de

terminar el diseño de las funciones en cuestión, puede copiar el contenido de tests.txt, pegarlo en el archivo en el que está trabajando y ejecutarlo. Si alguno de los casos de tests detecta errores, siguiendo el paso 6 de la receta, deberá realizar las modificaciones necesarias para solucionarlos.

Recuerde que el testing no constituye una garantía de correctitud. En 1970, Edsger W. Dijkstra dijo:

"*Program testing can be used to show the presence of bugs, but never to show their absence*"

Cuando un caso de test falla, estamos seguros de que el código presenta un problema. Sin embargo, en general, si el código pasa correctamente un cierto conjunto de casos de tests, no significa que sea correcto, sino, simplemente, que el mismo se comporta de manera esperada en dichos casos.

Se da una excepción a esto cuando el conjunto de los casos de tests coincide exactamente con el dominio en cuestión, lo cual muchas veces es imposible, dado que se trata de conjuntos muy grandes. Por ejemplo, puede testear la función `implica` en todo su dominio (dado que sólo se trata de 4 casos), pero no puede hacer lo mismo con la función `evaluar`.

De todas formas, el testing (bien usado) es una herramienta de gran utilidad a hora de producir código.

Observación. El contenido del archivo tests.txt **no** puede ser utilizado en los ejercicios 1 a 5.

2 Formato de entrega

- El trabajo práctico debe resolverse en grupos de dos integrantes. Cada persona puede participar en un único grupo. Un grupo puede estar integrado por personas de distinta comisión.
- Exactamente una persona por grupo debe realizar la entrega en el sitio, escribiendo en el campo "Comentarios" los apellidos y nombres de cada integrante.
- No se aceptarán entregas en las que no se haya seguido la receta para el diseño. Para que la entrega del trabajo práctico sea válida, todas las funciones deben contar con diseño de datos, signatura, declaración de propósito, casos de prueba (si corresponde) y código. En caso contrario, se considerará que el TP no fue entregado.
- La entrega consiste en un único archivo por grupo: el archivo TP6-Apellido1-Apellido2.rkt que deberá ser editado apropiadamente. En concreto, se solicita:
  - Completar los datos pedidos para cada integrante del grupo según se indica en el archivo.
  - Completar el archivo con la resolución.
  - Cambiar el nombre del archivo reemplazando Apellido1 y Apellido2 por los apellidos de los integrantes, en orden alfabético ascendente.