Instituto Tecnológico de Costa Rica
Escuela de Ingeniería en Computación
IC-4700 Lenguajes de programación
Asignación #4 (proyecto para aprender sobre procesamiento simbólico con lenguajes funcionales

Historial de revisiones:
- 2020.12.18: Versión base (v0), a partir de una especificación de 2020.07.

Lea con cuidado este documento. Si encuentra errores en el planteamiento[1], por favor comuníqueselos inmediatamente al profesor.

Objetivo
Al concluir esta asignación, Ud. habrá construido procesadores de expresiones simbólicas: un comprobador de tautologías sobre proposiciones lógicas con variables, un convertidor de tales proposiciones hacia la forma normal disyuntiva y un 'impresor' de proposiciones lógicas con variables. La programación deberá ser desarrollada en el lenguaje de programación *Haskell*.

Bases
El profesor compartirá con el grupo el código y las técnicas usadas para construir otros evaluadores de expresiones simbólicas: un evaluador de proposiciones lógicas con *constantes* (*sin* variables), evaluadores de expresiones aritméticas simples, con constantes y con variables, y un comprobador de tautologías para proposiciones lógicas con *variables*. Todos esos programas están escritos en el lenguaje *Standard ML* en un estilo funcional. Ocasionalmente se utilizan *excepciones* en esos programas.

Sintaxis abstracta del lenguaje de proposiciones lógicas
Vamos a trabajar con un lenguaje de fórmulas lógicas, o *proposiciones*, donde aparecen *variables proposicionales*, representadas por hileras (*string*s), las *constantes* `true` y `false`, y los *conectivos lógicos* usuales: negación, conjunción, disyunción, implicación y equivalencia (doble implicación).

En Standard ML, este es un `datatype` que podemos utilizar para codificar las proposiciones.

```
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
```

Ud. definirá el tipo de datos (`data`) del lenguaje Haskell que sea capaz de representar términos equivalentes a los del `datatype Proposicion`.

Entradas
Las proposiciones serán valores de un tipo de datos (`data`) de Haskell equivalente al `datatype Proposicion` (en Standard ML).

Requerimientos
Especificamos sucintamente los requerimientos de las funciones ya implementadas en el lenguaje Standard ML por el profesor.

[1] El profesor es un ser humano, falible como cualquiera.

1. Programar una función, `vars`, que determina la ***lista*** de las distintas *variables proposicionales* que aparecen en una fórmula lógica (proposición). La lista no debe tener elementos repetidos.
2. Programar una función, `gen_bools`, que produce todas las posibles combinaciones de valores booleanos para *n* variables proposicionales. Si hay *n* variables proposicionales, tendremos 2^n arreglos distintos de *n* valores booleanos.
3. Programar una función, `as_vals` que, dada una lista de variables proposicionales sin repeticiones, la combina con una lista de valores booleanos (`true` o `false`) de la misma longitud, para producir una lista del tipo `(string * bool) list` – en Haskell `[(String, Bool)]` – que combina, posicionalmente, cada variable proposicional con el correspondiente valor booleano. Esto lo denominamos una *asignación de valores* (a las variables proposicionales).
4. Definir una función, `evalProp`, que evalúa una proposición dada una *asignación* de valores booleanos a las variables proposicionales[2].
5. Definir una función, `taut`, que determina si una proposición lógica es una *tautología*, esto es, una fórmula lógica que evalúa a *verdadera* (`true`), para *toda* posible asignación de valores de verdad a las variables presentes en la fórmula.
   - Si la proposición lógica ***sí*** es una tautología, la función muestra la fórmula, seguida de la leyenda "es una tautología".
   - Si la proposición lógica ***no*** es una tautología, la función muestra la fórmula, seguida por la leyenda "no es una tautología" y muestra (al menos) una de las asignaciones de valores que produjeron `false` como resultado de la evaluación de la fórmula con esa asignación de valores.

   Por ejemplo,
   - $(p \vee \neg p)$ ***sí*** es una tautología.
   - $(q \Rightarrow \neg q)$ ***no*** es una tautología, porque q = `true` la *falsifica* (la hace falsa).
6. Definir una función, `fnd`, que obtiene la *forma normal disyuntiva* de una proposición lógica[3]
7. Definir una función, `bonita`, que genera una 'impresión' de la proposición lógica en la cual aparecen únicamente los paréntesis estrictamente necesarios. La decisión respecto de los paréntesis debe corresponderse con la jerarquía de precedencia entre los operadores (conectivos) lógicos. `bonita` debe imprimir primero la fórmula como un valor del tipo `Proposicion` y, línea aparte, la fórmula según las reglas descritas de seguido. La 'impresión' puede ser una hilera (`String`) de Haskell.

   Reglas de formato *básicas*:
   - Las constantes booleanas deben aparecer así: `True` y `False`.
   - Las variables deben aparecer *verbatim* (tal cual están escritas).
   - El operador de negación es unario y tiene la máxima precedencia.
   - El operador de implicación debe *asociar a la derecha*. Todos los demás operadores binarios deben asociar a la izquierda.
   - Los símbolos por usar son estos:
     - **~** para la negación, este es un símbolo unario y se aplica como un prefijo
     - **/** para la conjunción, este es un símbolo binario y se usa de manera infija
     - **\/** para la disyunción, es binario y se usa de manera infija
     - **=>** para la implicación, es binario y se usa de manera infija
     - **<=>** para la equivalencia lógica (doble implicación), es binario y se usa de manera infija
   - El orden de precedencia de los operadores lógicos es, de mayor a menor precedencia, como sigue:
     - **~** (negación)
     - **/** para la conjunción
     - **\/** para la disyunción
     - **=>** para la implicación
     - **<=>** para la equivalencia lógica (doble implicación)

---

[2] Revise la forma en que se evalúan las expresiones aritméticas cuando hay variables.
[3] Debe estudiar la forma normal disyuntiva. Se sugiere estudiar el algoritmo de Quine & McCluskey o el método de Blake.

Pruebas
Sus pruebas deben dar evidencia del adecuado funcionamiento de su programa y de los elementos que lo conforman.

Diseñe, ejecute y analice diversos casos de prueba para mostrar el comportamiento de las funciones básicas: `vars`, `gen_bools`, `as_vals`, `evalProp` y `taut`, así como cualquier función auxiliar que haya creado para construir las funciones principales. Proceda de manera semejante con las funciones `fnd` y `bonita`.

En una sección de su documentación, describa las pruebas y, en apéndice aparte, muestre las corridas de sus pruebas sobre el programa y sus elementos, con datos no triviales.

Documentación
En una sección inicial debe documentar clara y concisamente los siguientes puntos:

- Describir su estrategia para diseñar y construir la función `vars`, así como cualquier función auxiliar que forme parte de su estrategia.
- Describir su estrategia para diseñar y construir la función `gen_bools`, así como cualquier función auxiliar que forme parte de su estrategia.
- Describir su estrategia para diseñar y construir la función `as_vals`, así como cualquier función auxiliar requerida por su estrategia.
- Describir su estrategia para diseñar y construir la función `evalProp`, así como cualquier función auxiliar requerida por su estrategia.
- Describir su estrategia para diseñar y construir la función `taut`, así como cualquier función auxiliar que forme parte de su estrategia.
- Describir su estrategia para diseñar y construir la función `fnd`, así como cualquier función auxiliar que forme parte de su estrategia.
- Describir su estrategia para diseñar y construir la función `bonita`, así como cualquier función auxiliar requerida por su estrategia.

- Añadir secciones sobre estos aspectos:
  - Descripción de las pruebas realizadas.
  - Instrucciones para ejecutar el programa.
  - Descripción de los problemas encontrados y cualquier otra limitación que tuvieran. En caso de haber implementado parcialmente la asignación, pueden mostrar la ejecución de aquellas partes del programa que trabajan bien.
  - Discusión y análisis de los resultados obtenidos. Conclusiones a partir de esto.
  - Reflexión sobre la experiencia de trabajar con los lenguajes Standard ML y Haskell, así como con el paradigma de programación funcional.
  - Descripción resumida de las tareas realizadas por cada miembro del grupo de trabajo.
  - Apéndice con el código fuente su solución.
  - Apéndice con los detalles de las pruebas creadas para ejercitar su programa y evidencias de los resultados obtenidos.
  - Referencias. Los libros, revistas y sitios Web que utilizó durante la investigación y desarrollo de su proyecto. Citar toda fuente consultada.

Archivos por entregar

- Debe guardar su trabajo en una carpeta comprimida (formato **`.zip`**) según se indica abajo[4]. Esto debe incluir:
  - Documentación indicada arriba en un solo documento, con una portada donde aparezcan los nombres y números de carnet de los miembros del grupo. El documento debe estar en formato `.pdf`.
  - Código fuente de su solución a esta asignación (en una carpeta, si fuera necesario).

[4] **No use** formato **`.rar`**, porque es rechazado por el sistema de correo-e del TEC.

- Pruebas (en carpeta aparte): descripción de las pruebas, código creado para *probar* su solución, corridas de las pruebas y evidencias (archivos de texto y 'pantallazos').

Entrega
Fecha límite: **lunes 2021.01.25,** antes de las 23:55. No se recibirán trabajos después de la fecha y la hora indicadas.

Los grupos pueden ser de *hasta* **4** personas.

Debe enviar por correo-e el ***enlace***[5] a un archivo comprimido almacenado en la nube con todos los elementos de su solución a estas direcciones: `itrejos@itcr.ac.cr` y `joscaes12@gmail.com` (José Alfredo Campos Espinoza, nuestro Asistente).

El asunto (*subject*) debe ser:
`IC-4700 - Asignación` **`4`** – carnet + carnet + carnet + carnet.

Si su mensaje no tiene el asunto en la forma correcta, su trabajo será castigado con −10 puntos; podría darse el caso de que su proyecto no sea revisado del todo (y sea calificado con 0) sin responsabilidad alguna del profesor o del asistente (caso de que su mensaje fuera obviado por no tener el asunto apropiado). Si su mensaje no es legible (por cualquier motivo), o contiene un virus, o es entregado en formato **`.rar`**, la nota será 0.

La redacción y la ortografía deben ser correctas. La citación de sus fuentes de información debe ser acorde con los lineamientos de la Biblioteca del TEC. En la carpeta 'Referencias', bajo 'Documentos públicos de LENGUAJES DE PROGRAMACION GR 1', ver 'Citas y referencias'. La Biblioteca del TEC usa mucho las normas de la APA. El profesor prefiere las de la ACM, pero está bien si usan las de APA para su trabajo académico en el TEC.

El profesor tiene *altas expectativas* respecto de la calidad de los trabajos escritos y de la programación producidos por estudiantes universitarios de la carrera de Ingeniería en Computación del Tecnológico de Costa Rica. Los profesores esperamos que los estudiantes tomen en serio la comunicación profesional.

Ponderación
Este proyecto tiene un valor del 15% de la calificación del curso.

[5] Los sistemas de correo del TEC rechazan el envío o la recepción de carpetas comprimidas con componentes ejecutables. Suban su carpeta comprimida (en formato **`.zip`**) a algún 'lugar' en la nube y envíen el hipervínculo al profesor y a su asistente mediante un mensaje de correo con el formato indicado. Deben mantener la carpeta viva hasta 4 de febrero del 2021.