Instituto Tecnológico de Costa Rica
Escuela de Ingeniería en Computación
IC-4700 Lenguajes de programación
Asignación #2 [Proyecto]: procesamiento simbólico con lenguajes funcionales (Standard ML)

Historial de revisiones:
- 2022.10.05: Versión base (v0).

Lea con cuidado este documento (*incluidas las notas a los pies de página*)**.** Si encuentra errores en el planteamiento[1], por favor comuníqueselos inmediatamente al profesor.

Objetivo
Al concluir esta asignación, Ud. habrá modificado y extendido un procesador de expresiones simbólicas: con base en un comprobador de tautologías sobre proposiciones lógicas con variables, ustedes crearán 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 *Standard ML*.

Bases
El profesor ha compartido 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*. Los programas están escritos en el lenguaje *Standard ML* en un estilo funcional. Ocasionalmente se utilizan *excepciones* en esos programas. El comprobador de tautologías, escrito en Standard ML, se presenta en dos formas: con excepciones y sin excepciones.

Sintaxis abstracta del lenguaje de proposiciones lógicas con variables
Vamos a trabajar con un lenguaje de fórmulas lógicas, o *proposiciones*, donde aparecen *variables proposicionales*, representadas por hileras (*string*s) no vacías, 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 con variables.

```
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
```

Entradas
Las proposiciones serán valores del tipo de datos `Proposicion` (un `datatype` en Standard ML). Para facilitar el ingreso de expresiones de tipo `Proposicion`, definimos como *operadores* a las funciones que siguen:

```
nonfix ~:
val ~: = negacion

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

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

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

```
infixr 5 :=>:
val (op :=>:) = implicacion

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

La negación es una función *prefija*. Las demás son *infijas*. `infixr` indica asociatividad a la derecha. `infix` indica asociatividad a la izquierda. El dígito que sigue a `infixr` o a `infix` establece el orden de precedencia (prioridad).

Bases
Especificamos sucintamente lo que hacen las funciones ya implementadas en el lenguaje Standard ML por el profesor. Estas serán útiles para que ustedes desarrollen este proyecto.

1. La función `vars` determina la ***lista*** de las distintas *variables proposicionales* que aparecen en una fórmula lógica (proposición). La lista no debe tener elementos repetidos.
2. La función `gen_bools` produce todas las posibles combinaciones de valores booleanos para *n* variables proposicionales. Si hay *n* variables proposicionales, tendremos 2^n maneras distintas de combinar *n* valores booleanos.
3. La función `as_vals`, dada una lista de variables proposicionales sin repeticiones y una lista de valores booleanos (`true` o `false`) de la misma longitud, produce una lista del tipo `(string * bool) list` que combina, posicionalmente, cada variable proposicional con el correspondiente valor booleano. Esto lo denominamos una *asignación de valores* (a las variables proposicionales) y se corresponde con el concepto de *asociación* (*binding*) estudiado previamente en el curso.
4. La función `evalProp` evalúa una proposición en el *contexto* de una *asignación* de valores booleanos a las variables proposicionales[2]. El contexto (o *ambiente*) permite determinar el valor de cada variable dentro de una proposición.
5. La función `taut` 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).

Funciones por desarrollar en este proyecto
Ustedes deberán *diseñar*, *construir*, *probar* e *integrar* (al programa final) las siguientes funciones.

6. Función `bonita`, que debe generar 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, en *línea aparte*, la fórmula según las reglas descritas abajo. La 'impresión' puede ser una hilera (`string`) de Standard ML, o bien ser mostrada mediante la función `print` de Standard ML.

   Reglas de formato:

[2] Otro ejemplo análogo aparece en el intérprete que evalúa expresiones aritméticas con variables.

- 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 [asocia a la izquierda]
  - **||** para la disyunción, es binario y se usa de manera infija [asocia a la izquierda]
  - **=>** para la implicación, es binario y se usa de manera infija [asocia a la *derecha*]
  - **<=>** para la equivalencia lógica, es binario y se usa de manera infija [asocia a la izquierda]
- El orden de precedencia de los operadores lógicos es, de *mayor* a *menor* precedencia, como sigue:
  - **~** la negación tiene la máxima precedencia
  - **&&** conjunción
  - **||** disyunción
  - **=>** implicación
  - **<=>** equivalencia lógica (doble implicación)

Tome en consideración que algunos paréntesis serán necesarios, pues señalarán el agrupamiento de una sub-proposición con precedencia mayor que el conectivo que la domina.

7. Función `fnd`, que debe obtener la *forma normal disyuntiva* de una proposición lógica[3].

8. (*) **Grupos de 4 miembros**: Diseñar, construir y validar una función, **`hermosa`**, que genere *documentos* que incluyan texto con símbolos matemáticos. Los documentos serán archivos de texto, cuyo contenido estará expresado en uno de estos lenguajes de marcas: LaTeX (`.tex`) o HTML+MathML (`.html` o `.htm`). Tengan cuidado con las tipografías. Los documentos deben ser guardados persistentemente en archivos y poder ser abiertos desde un procesador de LaTeX o un navegador Web que soporte MathML. Noten que MathML no es soportado por todos los navegadores[4]. Noten que los documentos LaTeX pueden necesitar incluir alguna biblioteca donde estén los símbolos apropiados.
   - Las constantes booleanas deben aparecer así: `true` y `false`, en una tipografía ('fuente', 'font') monoespaciada (*monospaced*)[5].
   - Las variables deben aparecer en *cursiva* (*italics*), preferiblemente con espaciado proporcional.
   - 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.
   - Las precedencias de los operadores son las indicadas en el punto 6.
   - Los símbolos por usar son estos:
     - ¬ para la negación
     - ∧ para la conjunción
     - ∨ para la disyunción
     - ⇒ para la implicación
     - ⇔ para la equivalencia lógica (doble implicación)

Desarrollo

Diseñe, programe, valide y documente las funciones `bonita`, `fnd` y `hermosa`, así como cualquier función auxiliar que haya creado para construir cualquiera de ellas. Se recomienda usar como base el código en Standard ML suministrado por el profesor.

---

[3] Ustedes deben estudiar la forma normal disyuntiva. Se sugiere consultar la sección 1.5 del libro del Prof. Murillo sobre Matemática discreta.

[4] **¡Cuidado!** Ver: https://www.lambdatest.com/web-technologies/mathml

[5] Por ejemplo, en Windows: Courier, Courier New, Andale Mono, FreeMono, DejaVu Sans, Lucida Console. En LATEX, usar `\tt`. Investiguen qué se puede hacer en HTML.

Pruebas
Valide su programación con diversos casos de prueba, que permitan mostrar el comportamiento de las funciones desarrolladas por su grupo. Sus pruebas deben dar evidencia del adecuado funcionamiento de su programa y de los elementos que lo conforman.

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.

Recuerde que, para cualquier proposición *prop*, debe cumplirse que *prop* `<=>` `fnd` *prop* es una tautología.

Informe técnico
Deberán preparar un informe técnico que incluya:

- Portada que identifique a los autores del informe, con sus carnets.
- Introducción al informe: describir el objetivo de su trabajo y las secciones que lo componen.
- Secciones:
  - Funciones `vars`, `gen_bools`, `as_vals`, `evalProp` y `taut`.
    - Breve explicación de lo que hace cada función.
    - Estrategia para probar cada una de esas funciones.
    - Pruebas y resultados obtenidos con cada una de esas funciones.
  - Función `fnd`.
    - Estrategia para diseñar, construir y probar la función `fnd`.
    - Código de la función `fnd`.
    - Descripción de funciones auxiliares, si las hubiera.
    - Pruebas y resultados obtenidos.
    - Referencias consultadas, si las hubiera.
  - Función `bonita`.
    - Estrategia para diseñar, construir y probar la función `bonita`.
    - Código de la función `bonita`.
    - Descripción de funciones auxiliares, si las hubiera.
    - Pruebas y resultados obtenidos.
    - Referencias consultadas, si las hubiera.
  - Función `hermosa`.
    - Estrategia para diseñar, construir y probar la función `hermosa`.
    - Decisiones de diseño tomadas en cuanto al formato elegido para el texto matemático.
    - Manejo de archivos de texto en Standard ML.
    - Código de la función `hermosa`.
    - Descripción de funciones auxiliares, si las hubiera.
    - Pruebas y resultados obtenidos.
    - Referencias consultadas, si las hubiera.
  - Discusión y análisis de los resultados obtenidos, en general.
  - Conclusiones del grupo a partir de sus resultados.
  - 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 y discutir qué fue lo que no pudieron completar o que no funcionó correctamente.
  - Reflexión sobre la experiencia de trabajar con el lenguaje Standard ML, así como con el paradigma de programación funcional.
  - Descripción resumida de las tareas realizadas por cada miembro del grupo de trabajo.
- Referencias:
  - Los libros, revistas y sitios Web que utilizaron durante la investigación y el desarrollo de su proyecto. Citar toda fuente consultada, incluido el código facilitado por el profesor.
- Apéndices:
  - Instrucciones para ejecutar el programa.
  - Código fuente su solución [incluido el código facilitado por el profesor].
  - Descripción general de las pruebas diseñadas para validar su programa.
  - Detalles con la ejecución de las pruebas sobre su programa, con evidencias de los resultados obtenidos.

Archivos por entregar

- Guardar su trabajo en *una* carpeta comprimida (formato **.zip**)[6]. Esto debe incluir:
  - Informe técnico, en un solo documento, según se indicó arriba. El documento debe estar en formato .pdf.
  - Apéndices con código fuente de los programas desarrollados por su grupo.
  - Apéndices con ejemplos de las corridas de sus programas. Si generaron videos: súbanlos a algún espacio compartido en la Nube y pongan los hipervínculos de acceso en un archivo de texto o en un .pdf.
  - El código fuente, todas las funciones, propias y ajenas, en una sub-carpeta.
- Subir su carpeta comprimida a alguna nube, obtener un enlace de solo-lectura, para pasarlo al profesor y al asistente del curso. Deben mantener la carpeta accesible hasta 28 de febrero del 2023.

Entrega

Fecha límite: **miércoles 2022.10.26,** antes de las **23:45**.

Los grupos pueden ser de *hasta* **4** personas. El punto 8 es ***obligatorio*** para los grupos de 4 miembros.

Deben enviar por correo-e el ***enlace***[7] a un archivo comprimido almacenado en alguna nube, con todos los elementos de su solución a estas direcciones: itrejos@itcr.ac.cr, svengra01@gmail.com (Steven Granados Acuña, Asistente). El archivo comprimido debe llamarse **Asignación 2** *carnet carnet carnet*

El asunto (*subject*) de su mensaje debe ser: **IC-4700 Asignación 2** *carnet carnet carnet*

Sustituir *carnet* por los números de carnet de cada persona miembro del grupo.

Si su mensaje no tiene el asunto en la forma correcta, su trabajo será castigado con **-20** puntos; puede 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**.

Habilitaremos también la entrega de los trabajos vía el tecDigital.

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 (bibliografía), en OneDrive, ver 'Citas y referencias'. La Biblioteca del TEC usa mucho las normas de la APA. El profesor prefiere las de ACM; está bien si usan las normas bibliográficas de la ACM, la APA o el IEEE para su trabajo académico en el TEC (con este profesor).

El profesor tiene *altas expectativas* respecto de la calidad de los trabajos escritos y de la programación que produzcan 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.

Referencias

- [Morgan, 1994] Morgan, Carroll. *Programming from specifications, 2nd ed.* Prentice Hall International, 1994. Ver: *Appendix A: Some laws for predicate calculation*, particularmente la sección A.1
- [Murillo, 2010] Murillo Tisjli, Manuel. *Introducción a la Matemática Discreta, 4ta ed.* Editorial Tecnológica de Costa Rica, 2010. Ver: secciones *1.3 Leyes de la lógica* y *1.5 Formas normales*.

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

[7] Los sistemas de correo han estado rechazando 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 nuestro asistente mediante un mensaje de correo con el formato indicado. Deben mantener la carpeta accesible hasta 28 de febrero del 2023.

Estas referencias están en la carpeta ‘Lógica’[8], bajo ‘Referencias (bibliografía)’[9] en OneDrive.

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

Ejemplitos

```
- pru10 ;
> val it =
    disyuncion(disyuncion(variable "p",
                          conjuncion(negacion(variable "p"), variable "q")),
               negacion(variable "q")) : Proposicion

- bonita pru10;
> val it = "p || ~ p && q || ~ q" : string

- pru14;
> val it = conjuncion(variable "p", disyuncion(variable "p", variable "q")) :
  Proposicion

- bonita pru14 ;
> val it = "p && (p || q)" : string

- pru15 ;
> val it =
    conjuncion(disyuncion(variable "p", negacion(variable "p")),
               equivalencia(variable "q", negacion(variable "q"))) :
  Proposicion

- bonita pru15 ;
> val it = "(p || ~ p) && (q <=> ~ q)" : string
```

[8] https://tecnube1-my.sharepoint.com/personal/itrejos_itcr_ac_cr/_layouts/15/onedrive.aspx?ga=1&id=%2Fpersonal%2Fitrejos%5Fitcr%5Fac%5Fcr%2FDocuments%2FIC%2D4700%20Lenguajes%20de%20programaci%C3%B3n%2FReferencias%20%28bibliograf%C3%ADa%29%2FL%C3%B3gica
[9] https://tecnube1-my.sharepoint.com/personal/itrejos_itcr_ac_cr/_layouts/15/onedrive.aspx?id=%2Fpersonal%2Fitrejos%5Fitcr%5Fac%5Fcr%2FDocuments%2FIC%2D4700%20Lenguajes%20de%20programaci%C3%B3n%2FReferencias%20%28bibliograf%C3%ADa%29&ga=1