Instituto Tecnológico de Costa Rica
Escuela de Ingeniería en Computación
IC-4700 Lenguajes de programación

Asignación #2 [Proyecto], Procesamiento simbólico con lenguajes funcionales

Historial de revisiones:
- 2021.05.30: Versión base (v0).

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 reutilizado o construido procesadores de expresiones simbólicas: un comprobador de tautologías sobre proposiciones lógicas con variables, un convertidor de tales proposiciones hacia la forma normal disyuntiva, un simplificador de proposiciones, 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 utilizadas 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) 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.

```
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
```

Para facilitar el ingreso de expresiones de tipo `Proposicion`, definimos las funciones que siguen:

```
nonfix ~:
val ~: = negacion

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

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

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

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

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

Todas son *infijas*, menos la negación.

Entradas
Las proposiciones serán valores del `datatype Proposicion` (en Standard ML).

Requerimientos

Funciones base dadas por el profesor
Primero describimos sucintamente las funciones ya implementadas en el lenguaje Standard ML por el profesor.

1. Función `vars`, que determina la ***lista*** de las distintas *variables proposicionales* que aparecen en una fórmula lógica (proposición). La lista resultante no tiene elementos repetidos.
2. 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. 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` que combina, posicionalmente, cada variable proposicional con el correspondiente valor booleano. Esto lo denominamos una *asignación de valores* (a las variables proposicionales). [En clase también la hemos denominado una *asociación de variables a valores booleanos*.]
4. Función `evalProp`, que evalúa una proposición dada una *asignación* de valores booleanos a las variables proposicionales[2].
5. 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).

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

6. `fnd`, que obtiene la *forma normal disyuntiva* de una proposición lógica[3].
7. `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, en línea aparte, la fórmula según las reglas descritas adelante. La 'impresión' puede ser una hilera de Standard ML o ser impresa vía la función `print` (en Standard ML).

   Reglas de formato *básicas*:
   - Las constantes booleanas deben aparecer así: `true` y `false`.
   - Las variables deben aparecer *verbatim* (tal cual están escritas).

---

[2] Para facilitar la comprensión, puede serle útil revisar la forma en que se evalúan las expresiones aritméticas cuando hay variables.
[3] Debe estudiar la forma normal disyuntiva. Se sugiere consultar la sección 1.5 del libro de Murillo. Los aventurados también pueden estudiar el algoritmo de Quine & McCluskey o el método de Blake, que resuelven otros problemas relacionados.

- 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:
  - `~` para la negación
  - `/\` para la conjunción
  - `\/` para la disyunción
  - `=>` para la implicación
  - `<=>` para la equivalencia lógica (doble implicación)

8. Definir una función, `simpl`, que simplifica una proposición lógica reiteradamente hasta obtener una proposición lógica equivalente que no es posible simplificar más, según las reglas investigadas e implementadas por su grupo[4].
   - Grupos de **1** miembro: `simpl` debe ser capaz de aplicar al menos 5 reglas de simplificación.
   - Grupos de **2** miembros: `simpl` debe ser capaz de aplicar al menos 9 reglas de simplificación.
   - Grupos de **3** miembros: `simpl` debe ser capaz de aplicar al menos 13 reglas de simplificación.
   - Grupos de **4** miembros: `simpl` debe ser capaz de aplicar al menos 17 reglas de simplificación.

Pruebas

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

- Diseñe, documente, ejecute y analice diversos casos de prueba para mostrar el comportamiento de las funciones básicas dadas por el profesor: `vars`, `gen_bools`, `as_vals`, `evalProp` y `taut`.
- Diseñe, documente, ejecute y analice diversos casos de prueba para mostrar el comportamiento de las funciones desarrolladas por su grupo, así como cualquier función auxiliar que haya creado para construir las funciones `fnd`, `bonita` y `simpl`.

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:
  - Función `fnd`.
    - Estrategia para diseñar y construir 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 y construir 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 `simpl`.

[4] Este problema es retador. Vamos a discutir en clase sobre las dificultades que presenta el problema. Las referencias [Morgan, 1994] y [Murillo, 2010] documentan varias reglas de simplificación.

        - Estrategia para diseñar y construir la función `simpl`.
        - Código de la función `simpl`.
        - 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.
- Referencias:
    - Los libros, revistas y sitios Web que utilizaron durante la investigación y el desarrollo de su proyecto. Citar toda fuente consultada.
- Apéndices:
    - Descripción resumida de las tareas realizadas por cada miembro del grupo de trabajo.
    - 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 y con evidencias de los resultados obtenidos.

Archivos por entregar

- Deben guardar su trabajo en *una* carpeta comprimida (formato **`.zip`**) según se indica abajo[5]. Esto debe incluir:
    - Informe técnico, en un solo documento, según se indicó arriba. El documento debe estar en formato `.pdf`.
    - Carpeta con el código fuente de los programas desarrollados por su grupo.
    - Carpeta con evidencias de las pruebas realizadas.

Entrega

Fecha límite: **miércoles 2021.06.16,** 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***[6] a un archivo comprimido almacenado en la nube con todos los elementos de su solución a estas direcciones: `itrejos@itcr.ac.cr`, `susana.cob.3@gmail.com` (Susana Cob García, asistente), `a.tapia1908@gmail.com` (Alejandro Tapia Álvarez, asistente). El archivo comprimido debe llamarse `Asignación 3 carnet carnet carnet carnet`

El asunto (*subject*) de su mensaje debe ser:
`IC-4700: Asignación 3 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 de l@s asistentes (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.

---

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

[6] 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 nuestros asistentes mediante un mensaje de correo con el formato indicado. Deben mantener la carpeta viva hasta 15 de julio del 2021.

La redacción debe ser clara y la ortografía debe ser correcta. 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. Yo prefiero las de ACM, pero está bien si usan 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.

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*.

Estas referencias están en el tecDigital, carpeta 'Referencias' > 'Lógica'[7].

Ponderación

Este proyecto tiene un valor del 25% de la calificación del curso.

[7] https://tecdigital.tec.ac.cr/dotlrn/classes/CA/IC4700/S-1-2021.CA.IC4700.1/file-storage/#/102662823