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

Proyecto #1, procesadores para fórmulas lógicas con variables.

Historial de revisiones:
- 2020.06.15: Versión base (v0).
- 2020.06.17: v1.
- 2020.06.20: v2.1. Describir las extras.
- 2020.06.21: v2.2. Cambio en fecha de entrega.
- 2020.07.05: v3.0. Cambio en alcance básico. `:=>:` asocia a la derecha.

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 varios 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 simplificador de proposiciones.

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) y evaluadores de expresiones aritméticas simples, con constantes y con variables. Además, el profesor definirá el `datatype` (tipo de datos recursivo) de base para representar las proposiciones con variables (ver abajo).

Entradas

Las proposiciones serán ingresadas interactivamente en el ambiente de ejecución de Standard ML (Moscow ML u otro), aprovechando la definición de operadores lógicos infijos y prefijos facilitados por el profesor.

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).

Este es el `datatype` que deberán 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 trabajo con estas expresiones, definimos los operadores lógicos que siguen:

```
nonfix ~:
val ~: = negacion

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

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

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

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

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

Requerimientos

1. Definir 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. Definir una función, `evalProp`, que evalúa una proposición dada una *asignación* de valores booleanos a las variables proposicionales[2].
3. Defina 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.
4. Defina 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` que combina, posicionalmente, cada variable proposicional con el correspondiente valor booleano. Esto lo denominamos una *asignación de valores* (a las variables proposicionales).
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. **Extra**. Definir una función, `fnd`, que obtiene la *forma normal disyuntiva* de una proposición lógica. *Esto exige estudiar bien la forma normal disyuntiva. Se sugiere estudiar el algoritmo de Quine & McCluskey o el método de Blake.*
7. 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[3]. `simpl` debe ser capaz de aplicar al menos 7 reglas de simplificación.
8. **Extra**. `simpl` debe ser capaz de aplicar al menos 17 reglas de simplificación. Deben justificar las razones por las cuales escogieron las 17 reglas.
9. **Extra**. 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.

   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.

---

[2] Revise la forma en que se evalúan las expresiones aritméticas cuando hay variables.
[3] Este problema es particularmente difícil. Vamos a discutir en clase sobre las dificultades que presenta el problema.

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

Reglas de formato *para los aventurados*:
- Generar documentos .docx, LaTeX o .html. Elijan uno de los formatos. Tengan cuidado con las tipografías.
- Las constantes booleanas deben aparecer así: `true` y `false`, en una tipografía ('fuente', 'font') monoespaciada (*monospaced*)[4].
- Las variables deben aparecer en *cursiva* (*italics*).
- 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
  - ∧ para la conjunción
  - ∨ para la disyunción
  - ⇒ para la implicación
  - ⇔ para la equivalencia lógica (doble implicación)

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 principales: `vars`, `evalProp`, `gen_bools`, `as_vals`, `taut`, ~~`fnd`~~ y `simpl`, así como cualquier función auxiliar que haya creado para construir las funciones principales. Proceda de manera semejante si desarrolló las funciones `bonita` y `fnd`.

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 `evalProp`, así como cualquier función auxiliar requerida por 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 `taut`, así como cualquier función auxiliar que forme parte de su estrategia.
- Describir las reglas de simplificación investigadas por su grupo y las decisiones que tomaron respecto de cuáles implementar en su función `simpl`[5]. Justificar su elección de las reglas.

[4] Por ejemplo, en Windows: Courier, Courier New, Andale Mono, FreeMono, DejaVu Sans, Lucida Console. En LATEX, usar `\tt`.
[5] Recomiendo consultar el Apéndice A de [Morgan, 1994].

- Describir su estrategia para diseñar y construir la función `simpl`, así como cualquier función auxiliar requerida por su estrategia.
- **Para los que hacen lo extra**:
    - Describir su estrategia para diseñar y construir la función `bonita`, así como cualquier función auxiliar requerida por su estrategia.
    - Describir su estrategia para diseñar y construir la función `fnd`, así como cualquier función auxiliar que forme parte de su estrategia.

- **Para todos**: Añadir secciones sobre estos aspectos:
    - Descripción de las pruebas realizadas.
    - Instrucciones para ejecutar el programa.
    - **Estado** en que quedó su 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 el lenguaje Standard ML y 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[6]. 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).
    - 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 2020.07.13,** antes de las 23:55. No se recibirán trabajos después de la fecha y la hora indicadas.

Los grupos pueden ser de *hasta* **3** personas. Se admiten grupos de **4** miembros si hacen *todos* los requerimientos indicados como extra.

Debe enviar por correo-e el ***enlace***[7] a un archivo comprimido almacenado en la nube con todos los elementos de su solución a estas direcciones: `itrejos@itcr.ac.cr` y `andres.mirandaarias@gmail.com` (Andrés Miranda Arias, nuestro asistente).

> El asunto (subject) debe ser:
> `IC-4700 - Proyecto 1 -` carnet + carnet + carnet.
>
> Los carnets deben ir ordenados ascendentemente.

Si su mensaje no tiene el asunto en la forma correcta, su proyecto 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

---

[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 su asistente mediante un mensaje de correo con el formato indicado.

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. El profesor tiene *altas expectativas* respecto de la calidad de los trabajos escritos y de la programación producidos por estudiantes universitarios de tercer año 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, Carroll. *Programming from specifications, 2nd. Ed.* Prentice Hall International, 1994. Ver Appendix A: *Some laws for predicate calculation*.

Ponderación

Este proyecto tiene un valor del 30% de la calificación del curso.