Profesor: Federico Olmedo Ismael Figueroa
Auxiliares: Vicente Gatica Rodrigo Urrea Damian Arquez
Ayudantes: Cristobal Ardiles Rodrigo Iturrieta José Romero
Cristóbal Suazo Cristian Carrión Gaspar Ricci
Lucas Murray Rodrigo Llul

CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE CHILE

Tarea 1
Programación Funcional en Racket

Para la resolución de la tarea recuerde que

- Toda función debe estar acompañada de su firma, una breve descripción coloquial (en `T1.rkt`) y un conjunto significativo de tests (en `test.rkt`).
- Todo datatype definido por el usuario (via `deftype`) debe estar acompañado de una breve descripción coloquial y de la gramática BNF que lo genera.

Si la función o el datatype no cumple con estas reglas, será ignorado.

Ejercicio 1 **29 Pt**

Una *proposición* es una formula booleana con una de las siguientes formas:

- un nombre (string)
- $p \wedge q$
- $p \vee q$
- $\neg p$

donde p y q son proposiciones.
En particular, podemos definir el conjunto Prop usando las siguientes reglas:

$$\frac{n \in \mathsf{String}}{(\mathsf{varp}\ n) \in \mathsf{Prop}} \qquad \frac{p \in \mathsf{Prop} \qquad q \in \mathsf{Prop}}{(\mathsf{andp}\ p\ q) \in \mathsf{Prop}} \qquad \frac{p \in \mathsf{Prop} \qquad q \in \mathsf{Prop}}{(\mathsf{orp}\ p\ q) \in \mathsf{Prop}}$$

$$\frac{p \in \mathsf{Prop}}{(\mathsf{notp}\ p) \in \mathsf{Prop}}$$

Por ejemplo, la proposición $a \vee (b \wedge \neg c)$ se representa a través de

```
(orp (varp "a") (andp (varp "b") (notp (varp "c"))))
```

(a) [4 Pt] Defina el tipo de datos recursivo `Prop` y acompáñelo de su gramática.

(b) [5 Pt] Utilizando recursión explicita, defina la función

```
;; occurrences :: Prop String -> Number
```

que devuelve la cantidad de veces que una variable aparece en una proposición.

```
>>> (occurrences (varp "a") "b")
0
>>> (occurrences (andp (varp "a") (varp "b")) "a")
1
>>> (occurrences (andp (varp "a") (varp "a")) "a")
2
```

(c) [5 Pt] Utilizando recursión explicita, defina la función

```
;; vars ::  Prop -> (Listof String)
```

que devuelve una lista con todos los nombres de variables que ocurren en la proposición. La lista retornada no debe tener duplicados.

```
>>> (vars (varp "a"))
(list "a")
>>> (vars (andp (varp "a") (varp "b")))
(list "a" "b")
>>> (vars (andp (varp "a") (varp "a")))
(list "a")
```

Nota: Le puede resultar útil la función `remove-duplicates`.

(d) [5 Pt] Defina la función recursiva

```
;; all-environments ::  (Listof String) -> (Listof (Listof (Pair String
    Boolean)))
```

que dada una lista de variables (sin duplicados), crea todos los ambientes de evaluación posible

```
>>> (all-environments (list))
(list (list))
>>> (all-environments (list "a"))
(list
   (list (cons "a" #t))
   (list (cons "a" #f)))
>>> (all-environments (list "a" "b"))
(list (list (cons "a" #t) (cons "b" #t))
      (list (cons "a" #t) (cons "b" #f))
      (list (cons "a" #f) (cons "b" #t))
      (list (cons "a" #f) (cons "b" #f)))
```

(e) [5 Pt] Utilizando recursión explicita, defina la función

```
;; eval :: Prop (Listof (Pair String Boolean)) -> Boolean
```

que evalúa una proposición `p`, obteniendo los valores de cada variables desde una ambiente `env`, devolviendo el valor de verdad de dicha fórmula. Asuma que la lista no contiene dos veces una misma variable. En caso de que el nombre de una variable no aparezca en el ambiente, debe lanzar un error siguiendo la siguiente especificación:

```
>>> (eval (varp "a") (list (cons "a" #t)))
#t
>>> (eval (varp "a") (list (cons "a" #f)))
#f
>>> (eval (varp "a") (list))
eval: variable a is not defined in environment
```

Nota: Para realizar la búsqueda de un nombre en el ambiente, le puede ser útil la función `assoc`.

(f) [5 Pt] Defina la función recursiva

```
;; tautology? :: Prop -> Boolean
```

que retorna `#t` si la proposición es una tautología, es decir, si es verdadera para cualquier ambiente de evaluación (y `#f` en caso contrario).

```
>>> (tautology? (orp (varp "a") (notp (varp "a"))))
#t
>>> (tautology? (andp (varp "a") (notp (varp "a"))))
#f
```

Ejercicio 2 — 12 Pt

Una forma normal disyuntiva (o en inglés: *Disjunctive Normal Form (DNF)*) es una forma canónica de una fórmula lógica que se representa como una disyunción de conjunciones de literales, donde un literal es una variable proposicional o su negación, es decir, un *OR de ANDs de literales*. Puede ver la página de DNF en wikipedia para ver ejemplos de formulas en DNF.

En este ejercicio implementaremos un algoritmo para convertir cualquier proposición arbitraria en su forma DNF. Para ello seguiremos los siguientes pasos:

1. **Simplificar las negaciones**. Debemos eliminar todas las negaciones doble, y aplicar las leyes de De Morgan cada vez que podamos:

$$\begin{aligned} \neg\neg p &\mapsto p \\ \neg(p \wedge q) &\mapsto \neg p \vee \neg q \\ \neg(p \vee q) &\mapsto \neg p \wedge \neg q \end{aligned}$$

   Este paso se debe repetir hasta que no quede ninguna de las anteriores transformaciones disponibles.

2. **Distribuir todas las conjunciones (AND)**. Debemos aplicar las leyes de distributividad para AND cada vez que podamos:

$$\begin{aligned} p \wedge (q \vee r) &\mapsto (p \wedge q) \vee (p \wedge r) \\ (p \vee q) \wedge r &\mapsto (p \wedge r) \vee (q \wedge r) \end{aligned}$$

   Al igual que el paso anterior, este paso se debe repetir hasta que no quede ninguna de las anteriores transformaciones disponibles.

Una vez que hayamos ejecutado todas las transformaciones anteriores tendremos una proposición en su formal normal disyuntiva.

(a) [3 Pt] Utilizando recursión explicita, defina la función

```
;; simplify-negations :: Prop -> Prop
```

que aplica las transformaciones del paso (1). Su función solo debe recorrer la estructura de la proposición una vez, sin importar si todavía quedan transformaciones por realizar.

```
>>> (simplify-negations (notp (notp (varp "a"))))
(varp "a")

>>> (simplify-negations (notp (andp (varp "a") (varp "b"))))
(orp (notp (varp "a")) (notp (varp "b")))

>>> (simplify-negations (notp (orp (varp "a") (varp "b"))))
(andp (notp (varp "a")) (notp (varp "b")))

>>> (simplify-negations (notp (orp (notp (varp "a")) (varp "b"))))
(andp (notp (notp (varp "a"))) (notp (varp "b")))
```

Nota: En el último ejemplo se puede dar cuenta que al aplicar la ley de De Morgan, se introdujo una doble negación. Este es el comportamiento esperado, dado que luego aplicaremos esta función las veces que sea necesario.

(b) [3 Pt] Utilizando recursión explicita, defina la función

```
;; distribute-and :: Prop -> Prop
```

que aplica las transformaciones del paso (2). Su función solo debe recorrer la estructura de la proposición una vez, sin importar si todavía quedan transformaciones por realizar.

```
>>> (distribute-and (andp (orp (varp "a") (varp "b")) (varp "c")))
(orp (andp (varp "a") (varp "c")) (andp (varp "b") (varp "c")))

>>> (distribute-and (andp (varp "c") (orp (varp "a") (varp "b"))))
(orp (andp (varp "c") (varp "a")) (andp (varp "c") (varp "b")))
```

(c) [3 Pt] Defina la función recursiva

```
;; apply-until :: (a -> a) (a a -> Boolean) -> a -> a
```

que dada una función `f` y un predicado `p` retorna una nueva función. Esta nueva función, dado un elemento `x`, le aplica `f` hasta que el predicado `p` (aplicado a los últimos dos resultados) retorna `#t`.

```
>>> ((apply-until
        (\lambda (x) (/ x (add1 x)))
        (\lambda (x new-x) (<= (- x new-x) 0.1))) 1)
0.25
```

Nota: en este ejemplo, la aplicación continua de la función sobre el argumento (1), resulta en los siguientes valores: $1 \to 0.5 \to 0.333 \to 0.25$. El predicado se cumple al llegar a 0.25, dado que $0.333 - 0.25 \leq 0.1$.

(d) [3 Pt] Defina la función

```
;; DNF :: Prop -> Prop
```

que dada una proposición, le aplica las transformaciones ya definidas tantas veces sea necesario para lograr la forma normal disyuntiva.

```
>>> (DNF (andp (orp (varp "a") (varp "b")) (orp (varp "c") (varp "d"))))
(orp
    (orp (andp (varp "a") (varp "c"))
         (andp (varp "a") (varp "d")))
    (orp (andp (varp "b") (varp "c"))
         (andp (varp "b") (varp "d")))
)
```

Nota: Puede utilizar la funcion `equal?` para verificar que ya no es necesario seguir aplicando transformaciones.

Ejercicio 3 **19 Pt**

(a) [4 Pt] Defina la función

```
;; fold-prop :: (String -> a) (a a -> a) (a a -> a) (a -> a) -> Prop -> a
```

que captura el esquema de recursión asociado a `Prop`

(b) [15 Pt] Redefina las funciones `occurrences`, `vars`, `eval`, `simplify-negations` y `distribute-and` usando el `fold-prop` arriba definido (en vez de una recursión explícita). Si necesita definir una función como argumento del `fold-prop`, hágalo usando funciones anónimas.