UNIVERSIDAD NACIONAL AUTÓNOMA
DE MÉXICO

FACULTAD DE CIENCIAS

Manual de laboratorio para el curso de Estructuras Discretas

REPORTE DE ACTIVIDAD DOCENTE

QUE PARA OBTENER EL TÍTULO DE:

Licenciada en Ciencias de la Computación

PRESENTA:

Daniela Calderón Pérez

TUTOR

Dr. Favio Ezequiel Miranda Perea

1. Datos del alumno	1. Datos del alumno
Apellido paterno	Calderón
Apellido materno	Pérez
Nombre (s)	Daniela
Teléfono	55 26 34 61 00
Universidad Nacional Autónoma de México	Universidad Nacional Autónoma de México
Facultad de Ciencias	Facultad de Ciencias
Carrera	Ciencias de la Computación
Número de cuenta	311002209

2. Datos del tutor	2. Datos del tutor
Grado	Dr
Nombre(s)	Favio Ezequiel
Apellido paterno	Miranda
Apellido materno	Perea

3. Datos del sinodal 1	3. Datos del sinodal 1
Grado	Dr
Nombre(s)	José David
Apellido paterno	Flores
Apellido materno	Peñaloza

4. Datos del sinodal 2	4. Datos del sinodal 2
Grado	Dra
Nombre(s)	Lourdes del Carmen
Apellido paterno	González
Apellido materno	Huesca

5. Datos del sinodal 3	5. Datos del sinodal 3
Grado	MCIC
Nombre(s)	Odín Miguel
Apellido paterno	Escorza
Apellido materno	Soria

6. Datos del sinodal 4	6. Datos del sinodal 4
Grado	M. en C.
Nombre(s)	Araceli Liliana
Apellido paterno	Reyes
Apellido materno	Cabello

7. Datos del trabajo escrito	7. Datos del trabajo escrito
Título	Manual de laboratorio para el curso de Estructuras Discretas
Subtítulo	
Número de páginas	138
Año	2019

PROGRAMA DE APOYO A PROYECTOS PARA LA INNOVACIÓN Y MEJORAMIENTO DE LA ENSEÑANZA (PAPIME) DE LA

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO, EN EL MARCO DEL PROYECTO:

“Tópicos en Ciencia de la Computación Teórica”
(UNAM-PAPIME, PE102117)

Dedicado a
mi mamá y abuelo.

Agradecimientos

Quiero agradecer en primer lugar a mi madre por todo su amor, apoyo, esfuerzo y confianza, gracias a ti todos mis sueños y metas se han cumplido, eres mi mejor ejemplo y me siento afortunada de ser tu hija. Te quiero mucho.

A mi abuelo, por siempre quererme, cuidarme y estar conmigo durante mi infancia, sé que este logro significaría mucho para ti.

También a mi padre, a mis tios y primos, por siempre apoyarme y contribuir con este logro. A Karla, por ser parte de este proceso desde el principio hasta el fin. Les agradezco de corazón.

A Jonathan por su apoyo, paciencia y compresión en todo momento. Gracias por ser parte de esta aventura.

También agradezco al Dr. Favio Ezequiel Miranda Perea por aceptar ser mi tutor y después incluirme en sus proyectos. Agradezo la paciencia que me tuvo, fue un gusto trabajar con usted.

A mis sinodales, Dr. José David Flores Peñaloza, Dra. Lourdes del Carmen González Huesca, MCIC. Odín Miguel Escorza Soria y M. en C. Araceli Liliana Reyes Cabello, quiero agradecerles su apoyo, consejos y el tiempo que le han dedicado a mi trabajo.

A todos los maestros que fueron parte de mi formación, les agradezco su esfuerzo y dedicación lo cual espero se vea reflejado, no sólo en este trabajo sino en todo mi desarrollo académico y profesional. Agradezco de manera especial a la profesora Guadalupe Ibargüengoitia, gracias por todos sus consejos y apoyo dentro y fuera de las aulas.

Agradezco a las profesoras y profesores que me permitieron ser su ayudande, vivir esa experiencia me enseñó mucho y contribuyó en el desarrollo de este trabajo. A mis alumnos, pues apredí de ustedes más de lo que yo pude enseñarles.

Por último, aunque no menos importante, agradezco a todos mis amigos por las risas, aventuras, consejos, etc., los quiero mucho. Y especialmente agradezco a Luis y Ricardo (Q.E.D), porque junto con Jonathan fuimos el *dream team*, gracias.

Índice general

Introducción

Objetivos generales

La materia de Estructuras Discretas cuenta con dos horas de laboratorio a la semana, al ser una materia de primer semestre, dicho laboratorio junto con el de la materia de Introducción a Ciencias de la Computación son el primer acercamiento que tienen los estudiantes con la programación dentro de la carrera y este laboratorio específicamente es el primer acercamiento con la programación funcional.

En las prácticas de este manual se relacionan temas de la teoría con ejercicios de programación, para ello utilizaremos *Haskell*, un lenguaje de programación que nos permite implementar los ejercicios con mayor facilidad gracias a sus características funcionales y también en ocaciones la sintaxis es similar a la notación utilizada en diferentes fuentes de la bibliografía. Además al ser uno de los lenguajes más populares en el paradigma funcional cuenta con bastante soporte, por lo que su instalación y uso no representarán un problema para los alumnos.

Relación teoría - práctica

Este manual esta dividido en dos partes, la primera parte es una introducción al lenguaje de programación, este acercamiento a *Haskell* consta de seís capítulos y cada uno abordará un tema básico con los que tendrán las bases para poder resolver las prácticas.
La segunda parte consta de prácticas relacionadas con la matería, por lo que cada una abordará algún punto del temario oficial.

En un semestre regular se cuentan con 16 semanas de clase, es decir, 16 clases de laboratorio de dos horas cada una, considerando el contenido de este manual, en las primeras seis clases se podrán ver los seis temas de *Haskell* y en las once clases restantes se podrán realizar las prácticas, las cuales están consideradas para resolverse en una semana.

Por lo tanto la distribución de las prácticas siguiendo el orden del temario oficial es el siguiente:

Unidad	**Tema**	**Práctica**
I Introducción	Gramáticas	Grámaticas
II Lógica matemática	Lógica proposicional	Tablas de verdad
	Tableaux semánticos	Tableaux
	Minimización de funciones booleanas	Mapas de Karnaugh
III Inducción y recursión	Recursión	Listas de longitud par
		Multiconjuntos
		Conversiones numéricas
		Permutaciones de listas
		Notación infija, sufija, prefija
IV Relaciones	Clases de equivalencia	Enteros y racionales como pares
	Conjuntos	Conjuntos como funciones

Estructura de las prácticas

Para facilitar el entendimiento de las prácticas, todas cuentan con un formato general que está dado por los siguientes puntos:

- **Objetivos**
  Se explican cuáles son los aprendizajes esperados y la relación que tiene la práctica con la teoría, Además se especifican los conocimientos previos requeridos tanto teóricos como pŕacticos para la realización de la práctica.
- **Preliminares**
  Es un breve repaso teórico del tema.
- **Implementación**
  Se explica la relación de la teoría con la práctica, los tipos que se utilizarán, cómo funcionan y algunos ejemplos.
- **Ejercicios**
  Se enlistan los ejercicios obligatorios de la práctica, al igual que las prácticas los ejercicios cuentan con un formato general dado por los siguientes puntos:
  - Nombre de la función
  - Firma
  - Especificación (Qué hacer en cada ejercicio)
  - Ejemplos
- **Extras**
  Ejercicios con un nivel de dificultad superior, que no se consideran obligatorios para el desarrollo satisfactorio de la práctica.
- **Cuestionario**
  Preguntas de teoría o práctica relacionadas con el tema.

Recomendaciones en la aplicación del manual

Dado que recursión es un tema sumamente importante para el uso de *Haskell* y la resolución de las prácticas, pero es un tema que usualmente se ve a mediados del semestre se sugiere una reestructuración en el orden del temario y por lo tanto un nuevo orden en la elaboración de las prácticas, quedando de la siguiente manera:

Unidad	**Tema**	**Práctica**
I Inducción y recursión	Recursión	Listas de longitud par
		Multiconjuntos
		Conversiones numéricas
		Permutaciones de listas
		Notación infija, sufija, prefija
II Relaciones	Clases de equivalencia	Enteros y racionales como pares
	Conjuntos	Conjuntos como funciones
III Introducción a los lenguajes formales	Gramáticas	Grámaticas
IV Lógica matemática	Lógica proposicional	Tablas de verdad
	Tableaux semánticos	Tableaux
	Minimización de funciones booleanas	Mapas de Karnaugh

De esta manera el tema de recursión se verá con más cercanía en el laboratorio y en la clase de teoría, y los ejercicios de laboratorio permitirían la apropiación del conocimiento en un tema de suma importancia en programación, además permitirá que el alumno practique recursión por más tiempo durante el semestre.

También la modificación del temario permite que prácticas que pueden considerarse más sencillas sean las primeras que el alumno resuelva, y prácticas que requieren de un nivel de abstracción mayor se resuelvan en un punto más avanzado del curso.

La parte introductoria del lenguaje de programación permanece en el orden establecido en el manual, y se siguen contemplando las seis primeras clases para su aplicación en el labortario.

PARTE I

HASKELL

Introducción a Haskell

Para conocer el lenguaje con el que trabajaremos, daremos un breve repaso de qué son los lenguajes de programación, para después adentrarnos en conocer el lenguaje que se manejará en el curso.

¿Qué es un lenguaje de programación?

Un lenguaje de programación es esencialmente un sistema notacional para representar cómputos en forma legible tanto para humanos como para computadoras.[2]

Clasificación de lenguajes de programación

A continuación haremos una breve descripción de la clasificación de los lenguajes de programación y explicaremos en qué consiste cada una de ellas:

1. **Nivel de abstracción** :
   - **Bajo nivel** :
     Son instrucciones que la computadora puede ejecutar directamente, como el *lenguaje de máquina* y el *lenguaje ensamblador*, pero estas instrucciones son específicas para cada tipo de procesador.
   - **Alto nivel** :
     Las instrucciones son más amigables para el razonamiento del programador, en particular, no tenemos que fijarnos en la estructura interna de la computadora.
2. **Paradigma** o estilo en que se diseñan las soluciones, en esta clasificación entran muchas categorías. A continuación se mencionan las más relevantes para el curso:
   - **Lenguajes imperativos** :
     Se basa en instrucciones que solucionan un problema de manera secuencial. Por lo que todas las instrucciones requieren de un orden de ejecución.
     Por ejemplo, para preparar limonada, tendríamos que especificar paso a paso, desde tomar los limones, preparar la jarra, cortar los limones, exprimirlos dentro de la jarra, etc.

- **Lenguajes Declarativos** :
  Aquí se indica que es lo que se está buscando o deseamos obtener, se describe el problema, especificando condiciones, proposiciones, afirmaciones, ecuaciones o transformaciones.
  En el ejemplo del agua de limón, en los lenguajes declarativos describiríamos el problema: si tenemos los elementos para preparar el agua de limón como precondición, se mezclan y se sirven, obtendremos agua de limón. Notemos que no está dicho cómo ejecutar cada paso de este proceso.

Los dos paradigmas que mencionamos, son los más generales, dentro de estos están incluidos paradigmas más específicos, ejemplos de ellos son:

- **Lenguajes lógicos** :
  Están incluidos en el paradigma declarativo, y su característica principal es la utilización de declaraciones lógicas que especifiquen las características que debe tener la solución buscada. [2]
- **Lenguajes orientados a objetos** :
  Son una extensión del paradigma imperativo, cuya característica principal es crear un sistema de clases y objetos.
- **Lenguajes funcionales** :
  Su característica principal es el uso de funciones que se combinan mediante composición de forma compleja para construir nuevas funciones. [2]

  En los lenguajes funcionales las estructuras de control de flujo como `for`, `while` y `do-while` no existen. Todo se procesa usando composición de funciones, recursividad, funciones de orden superior y currificación.

  Esto se debe a los fundamentos matemáticos de la mayoría de los lenguajes funcionales, principalmente con bases en el sistema formal diseñado por Alonzo Church para definir cómputos y estudiar las aplicaciones de las funciones llamado Cálculo Lambda.

  Hacemos énfasis en este paradigma, pues a él pertenece el lenguaje que se utilizará en el curso.

Funciones

Para entender mejor el comportamiento de los lenguajes funcionales, repasaremos lo que es una función.

Definición

Desde el punto de vista matemático, una función f es una regla de correspondencia que asocia, a cada elemento de un conjunto A determinado, con un único elemento de un segundo conjunto

B. [8] Usualmente nombramos al tipo A dominio y al tipo B contradominio.

Notación

Lo descrito anteriormente se expresa mediante: $f : A \to B$, la función de nombre f, tiene de dominio al conjunto A y de contradominio al conjunto B.
La imagen o aplicación del elemento $a \in A$ denotada por $f(a)$, es el único elemento $b \in B$ que le corresponde bajo $f : A \to B$.

Funciones de múltiples argumentos

Como hemos visto hasta ahora para tener una función necesitamos de un conjunto A (dominio) y un conjunto B (contradominio) y aplicamos la función f a un elemento $a \in A$ obteniendo un único elemento $b \in B$. Pero, ¿qué sucede si queremos aplicar nuestra función a más de un elemento que incluso pueden ser de diferentes conjuntos?. Usaremos entonces el producto cartesiano $A_1 \times A_2 \times A_3 \ldots \times A_n$, obteniendo así un nuevo conjunto de tuplas $(a_1, a_2, a_3 \ldots a_n)$, donde $a_1 \in A_1$, $a_2 \in A_2$, $a_3 \in A_3$, etcétera.
Por lo tanto las funciones de múltiples argumentos son de la forma $f : (A_1 \times A_2 \times A_3 \ldots \times A_n) \to B$, usando esta notación $f(a_1, a_2, a_3 \ldots a_n)$, indicando así que f recibe múltiples argumentos agrupados en una tupla.

Haskell

Contemplando los conceptos anteriores podemos adentrarnos en su relación con el lenguaje de programación *Haskell*.

Haskell es un lenguaje de programación funcional, es decir, todo se hace por medio de funciones bastante parecidas a las que vimos anteriormente. Las funciones en *Haskell* también cuentan con un dominio y contradominio, a los elementos del domino les llamaremos *argumentos* y el resultado de la función será el único elemento del contradominio que le corresponda a dichos argumentos.
Por lo tanto una función toma uno o más argumentos, produciendo un sólo resultado. Y su estructura general está dada por:

1. **Firma de la función** :

   ```
   nombreDeLaFuncion :: (T1, T2, T3, ... , T(n-1)) -> Tn
   ```

   donde cada T_i corresponde a un tipo de *Haskell*, los cuales veremos a detalle posteriormente.

2. **Cuerpo de la función** :
   Se escribe nuevamente el nombre de la funcion seguido de los nombres que tendrán los argumentos y después especificamos que hacemos con ellos.

   ```
   nombreDeLaFuncion (t1, t2, t3,..., t(n-1)) = especificación
   ```

Justamente en la especificación es donde definimos la regla de correspondencia de nuestra función.
Por lo tanto, una función en *Haskell* se ve de la siguiente manera:

```
nombreDeLaFuncion :: (T1, T2, T3, ... ,T(n-1)) -> Tn
nombreDeLaFuncion (t1, t2, t3 ,... , t(n-1)) = especificacion
```

Aunque la definición anterior es correcta, en *Haskell* se utilizan funciones *currificadas*. La *currificación* permite reemplazar argumentos estructurados como las tuplas, en una secuencia de argumentos más simples[8], por ejemplo:

```
suma :: (T1,T1) -> T1
suma (t1,t2) = t1 + t2
```

La función anterior representa la suma de dos elementos suponiendo que T_1 es un tipo numérico, si currificamos esta función obtendríamos la siguiente función:

```
sumaCurry :: T1 -> T1 -> T1
sumaCurry t1 t2 = t1 + t2
```

En ambas funciones la especificación es la misma, lo único que cambia es la manera en que se reciben los argumentos. En la primera función el argumento es un par, dicho par contiene los números a sumar, mientras que en la segunda función, se esperan dos argumentos donde cada uno represente uno de los números a sumar.

De manera general, una función currificada se ve de la siguiente manera:

```
nombreDeLaFuncion :: T1 -> T2 -> T3 ... -> T(n-1) -> Tn
nombreDeLaFuncion t1 t2 t3 ... t(n-1) = especificacion
```

Donde siempre el último tipo de la firma es decir T_n, es el tipo del resultado o contradominio, mientras que todos los tipos anteriores corresponden a argumentos de la función, es decir, a su dominio, esto porque en las funciones currificadas los tipos se asocian a la izquierda y por lo tanto no hay la necesidad de poner paréntesis a menos que queramos indicar un comportamiento específico de la función, como cuando tenemos funciones de orden superior, pero eso lo veremos más adelante.

Por convención de *Haskell* y simplicidad, las funciones que se verán y realizarán en este manual serán funciones currificadas.

Ejemplo :
Si queremos definir una función que devuelva el cuadrado de un número entero, determinamos primero su firma: `cuadrado :: Int -> Int` .
En la especificación debemos definir la regla de correspondencia de la función, en este caso, multiplicar por sí mismo el número de entrada. Entonces, el cuerpo de nuestra función queda así: `cuadrado x = x*x`.
Por lo tanto nuestra función cuadrado queda de la siguiente manera:

```
cuadrado :: Int -> Int
cuadrado x = x * x
```

Donde :
Int es el tipo primitivo de Enteros en *Haskell*.
x es el nombre con el que se identificará nuestro argumento.
x*x es el cálculo del resultado (especificación de la función).

Como mencionamos anteriormente, *Haskell* es un lenguaje de programación funcional, entonces, además de las características que nos brinda este paradigma, *Haskell* tiene otras peculiaridades.

- **Evaluación perezosa** :
  Haskell no ejecutará funciones ni calculará resultados hasta que se vea realmente forzado a hacerlo.
- **Tipado estático** :
  Cuando compilamos un programa, el compilador sabe qué tipo tiene cada pedazo de código.
- **Inferencia de tipos** :
  Esto significa que no tenemos que etiquetar cada trozo de código explícitamente con un tipo porque el sistema de tipos lo puede deducir de forma inteligente.
- **Polimorfismo** :
  Se puede definir una función para más de un tipo, por ejemplo, para funciones aritméticas básicas como la suma, resta y multiplicación existe una sóla función capaz de realizar la operación sin importar de que tipo numérico sean sus argumentos.
- **Casamiento de patrones (pattern matching)** :
  Consiste en una especificación de pautas que deben ser seguidas por los datos, los cuales pueden ser deconstruidos permitiéndonos acceder a sus componentes.

Estas y otras características se verán con más detalle más adelante.

¿Interpretado o compilado?

- GHC :
  Es un compilador de *Haskell*, puede generar código en C. Además cuenta con un ambiente interactivo *ghci*, que soporta la carga interactiva de código compilado.
  Para los fines de este curso, utilizaremos únicamente el ambiente interactivo *ghci*.

Instalación

La forma de instalar *Haskell* y GHC depende del sistema operativo con el que trabajemos, a continuación se muestra la manera más sencilla de instalar *Haskell* a partir de los sistemas operativos más comunes.

1. Ubuntu/Debian :
   $ sudo apt-get install haskell-platform

2. Fedora :
   $ sudo dnf install haskell-platform
3. OS X :
   https://www.haskell.org/platform/mac.html
4. Windows :
   https://www.haskell.org/platform/windows.html

Para la realización de todas las actividades de este manual no se solicitará trabajar con un sistema operativo en específico, aunque los ejemplos de uso del ambiente interactivo estarán hechos en *Linux*.

Tipos y clases

Manejo de Haskell y su intérprete

Ya vimos de forma muy general qué es *Haskell* y cuáles son sus características, ahora vamos a ver cómo utilizarlo.

Archivos de Haskell

Así como los archivos que usamos siempre en la computadora tienen un nombre y una extensión, por ejemplo los archivos PDF se ven de la forma nombre.pdf, los archivos de *Haskell* se nombrarán como cualquier archivo y su extensión será **.hs**, por ejemplo: holaMundo.hs.
Es una buena práctica de programación en *Haskell* que todas nuestras funciones se encuentren dentro de un módulo [1], [7] en nuestro caso cada práctica consistirá en un módulo. La definición de los módulos puede ser de dos formas:

1. Poner al principio del archivo

```
module Ejemplo where
```

   Donde Ejemplo es el nombre del módulo, `module` y `where` son palabras reservadas de *Haskell*.

2. Al igual que en el caso anterior nombramos al módulo, pero también indicamos todas las funciones que están definidas en él.

```
module Ejemplo
( fun1
, fun2
, fun3
, fun4
, fun5
, fun6
) where
```

[1] Colección de funciones, tipos y clases de tipos relacionadas entre sí.

En este caso, todas las funciones de nuestro módulo estan representadas por fun_n

Intérprete

Aunque *GHC* nos permite compilar un archivo *Haskell*, para los fines de este curso bastará con usar el ambiente interactivo *ghci* para probar nuestras funciones.
Una vez creado nuestro archivo e implementadas nuestras funciones hay dos maneras de poder interpretarlas. [2]

1. Estando en la terminal, posicionados en el directorio donde creamos nuestro archivo de *Haskell* (Documentos, Escritorio, etc.) escribimos: **ghci nombre_archivo.hs**.
   De esta forma entramos al ambiente interactivo de *Haskell* interpretando directamente nuestro módulo.

Figura 1: Interpretación de un archivo de *Haskell* desde la terminal *Linux*

2. Primero entrar al modo interactivo de *Haskell* escribiendo en terminal **ghci**, una vez en el ambiente interactivo escribimos : **:load nombre_archivo.hs**

[2] Los ejemplos están hechos para sistemas operativos con base *Linux*, el uso del intérprete puede variar dependiendo del sistema operativo.

Figura 2: Interpretación de un archivo de *Haskell* usando `load` desde una terminal *Linux*

En este ejemplo únicamente escribimos el nombre del archivo después de `load` pues el archivo se encuentra en el mismo directorio en el que estamos, pero si queremos interpretar un archivo de un directorio diferente debemos preceder al nombre del archivo con la ruta relativa a partir del directorio en que nos encontramos, por ejemplo:

Figura 3: Interpretación de un archivo de *Haskell* usando `load` desde un directorio diferente en terminal *Linux*

Una vez interpretado correctamente nuestro módulo podemos usar todas las funciones definidas en él, escribiendo el nombre de la función y los argumentos que necesita esta para funcionar.

Además de modularizar nuestros archivos otras convenciones de *Haskell* son:

- **Comentar nuestro código.**
  En *Haskell* podemos hacer dos clases de comentarios.

```
{-
  Comentario 1. Se usa usualmente para comentarios de mas
                de un renglon en nuestro codigo.
-}

--Comentario 2. Se usa usualmente para comentarios breves.
```

- **Nombre de los tipos, módulos y estructuras.**
  El nombre de todos los tipos en *Haskell* empiezan siempre con mayúsculas siguiendo el estandar *UperCamelCase*.

- **Nombre de las funciones y variables.**
  El nombre de las variables y funciones se debe procurar que sea corto y descriptivo, en caso de requerir un nombre de más de una palabra se debe seguir el estandar *lowerCamelCase*.

- Consultar Haskell.org para mayores referencias.

Comandos útiles dentro del intérprete

A continuación enlistamos los comandos más usados dentro del intérprete de Haskell.

- **:r** Utilizamos este comando cuando queremos reinterpretar un módulo.

- **:q** Utilizamos este comando para salir del intérprete.

- **:t** Utilizamos este comando para saber el tipo de una función, por ejemplo: *:t mod*.

- **:!clear** Utilizamos este comando para limpiar la consola.

- **:help** Para más comandos.

Preludio de Haskell

Como se puede ver en la Figura 2, cuando entramos primero al ambiente interactivo de *Haskell* se puede ver en terminal la etiqueta *prelude* >, esta indica que nos encontramos en el ambiente básico de *Haskell* y dicho ambiente cuenta ya con funciones definidas; a este conjunto de funciones se le conoce como *Preludio (Prelude)*, cuando ya interpretamos nuestro módulo podemos usar tanto las funciones definidas en el módulo como las que se encuentran ya definidas en el *preludio*.

Tipos básicos

Un tipo es una etiqueta que posee cada expresión en nuestro código. Esta etiqueta nos dice a que categoría de objetos se ajusta la expresión. [7]

Los tipos básicos con los que cuenta *Haskell* se mencionan a continuación:

- **Bool - Valores lógicos:**
  Este tipo contiene los dos valores lógicos **True** y **False**
- **Char - Caracteres individuales:**
  Este tipo contiene todos los caracteres individuales disponibles en un teclado normal. Ejem. 'A', 'b', '3'. Aunque también incluye representaciones para caracteres especiales como salto de línea('\n') y tabulador ('\t')
- **String - Cadenas de caracteres:**
  Este tipo contiene todas las secuencias de caracteres, como "abc", "1+1=2" y la cadena vacía " ".
- **Int - Enteros de tamaño fijo:**
  Este tipo contiene enteros como -100, 9, 1434. Podemos representar enteros entre el rango $[-2^{29}$ ... $2^{29-1}]$. Para enteros que excedan este rango, se pueden presentar resultados inesperados.
- **Integer - Enteros de tamaño arbitrario:**
  Este tipo contiene todos los enteros, con toda la memoria necesaria para su almacenamiento.
- **Float - Números de punto flotante:**
  Contiene números con punto flotante de precisión simple, es decir, números con punto decimal de tamaño fijo en memoria.
- **Double - Números de punto flotante:**
  Contiene números con punto flotante de precisión doble.

Tipo lista

Otro tipo de mucha importancia en *Haskell*, es el tipo lista, considerada en otros lenguajes como una estructura de datos, en *Haskell* se maneja como un tipo primitivo.

Definición del tipo lista

Diremos que una *lista* es una secuencia de elementos del mismo tipo. Dichos elementos se encuentran entre corchetes y separados por comas.
Ejemplos :

- $[1,2,3,4]$, una lista de enteros

- [”*hola*”,”*como*”,”*estas*”] , una lista de strings
- $[[1,2,3,4],[3,4,4,2,3,4],[5,6,4,5,4,2,]]$, una lista de listas de enteros

Algunas observaciones importantes de las listas son:

1. Dado un tipo T denotamos con [T] al tipo de listas cuyos elementos son del tipo T.
2. El número de elementos de una lista se denomina longitud (length).
3. La lista cuya longitud es cero es llamada lista vacía (empty o nil) y se denota con: [ ].
4. Una lista con solo un elemento $[t_1]$, es llamada lista unitaria (singleton list).
5. Es importante recalcar que las listas [] y [[]] son diferentes, la primera es una lista vacía, la segunda es una lista cuyo único elemento es la lista vacía.

Tipo tupla

Otro tipo primitivo de Haskell son las tuplas, las cuales son una secuencia finita de componentes cuyos tipos pueden ser diferentes. Sus componentes están entre paréntesis y separados por comas.
Ejemplos:

- $(8,$”*hola*”$)$
- (3.14,[1,2,3],”hola”)

Algunas observaciones acerca de las tuplas son:

1. Dados los tipos de $T_1 ... T_n$, el tipo tupla correspondiente se denota como $(T_1, ..., T_n)$
2. El número de componentes de una tupla se llama aridad (arity).
3. La tupla de aridad cero es la tupla vacía ().
4. Las tuplas de aridad uno no estan permitidas, pues podrian causar un conflicto con los paréntesis utilizados para hacer una evaluación en un orden explícito.
5. Los pares (tuplas de aridad 2), cuentan con dos operaciones predefinidas muy importantes:
   - `fst`: Obtiene el primer elemento de un par.
   - `snd`: Obtiene el segundo elemento de un par

Para tuplas con aridad mayor a dos, no existen funciones predefinidas, la única forma de obtener los elementos, es usando el casamiento de patrones *(pattern matching)*, como se verá más adelante.

Tipo función

Las funciones de *Haskell* que vimos en el capítulo anterior también son consideradas un tipo primitivo del lenguaje.

Variables de tipo

Cuando trabajamos con listas, una de las funciones más utilizadas es `append` denotada por (++). La especificación de esta función es concatenar dos listas, es decir, juntar dos listas en una sola. Como las listas no son heterogéneas, siempre que usemos `append`, esperamos trabajar con listas del mismo tipo, por ejemplo:

```
--Ejemplo 1.
Prelude> [1,2,3] ++ [4, 5, 6, 7]
[1,2,3,4,5,6,7]

--Ejemplo 2.
Prelude> [1.2,3.4,5.6] ++ [7.8,9.0] ++ [10.2,11.3,12.44]
[1.2,3.4,5.6,7.8,9.0,10.2,11.3,12.44]

--Ejemplo 3.
Prelude> "hola" ++ "como" ++ "estas" ++ "?"
"holacomoestas?"
```

En el Ejemplo 1 estamos concatenando listas de tipo Int, en el Ejemplo 2 listas de tipo Float, y en el Ejemplo 3 concatenamos listas de tipo Char, sin embargo, usamos la misma función para concatenar cualquier tipo de lista. ¿Habrá una definición diferente de `append`, para listas de cada tipo de datos?. No, hacer una función para cada tipo de datos no es una solución, por eso en Haskell se tienen las funciones polimórficas, es decir, una única función que está definida para más de un tipo. Para declarar esta clase de funciones, utilizaremos las variables de tipo.

Ejemplo:
Las funciones `fst` y `snd`, son funciones polimórficas, pues los pares pueden ser de cualquier tipo, la declaración de `fst` es la siguiente :

```
fst :: (a,b) -> a
fst (x,y) = x
```

Aquí (a,b) representa al tipo Tupla, *a* y *b* son *variables de tipo*, dado que las tuplas pueden ser de cualquier tipo, *x*, *y* son los nombres locales de las variables con las que identificaremos a los valores de tipo *a* y *b* respectivamente.

¿Qué pasa si queremos implementar una función que calcule el área de un rectángulo, sabiendo que la base y la altura podrían ser enteros o decimales?, ¿sería correcto hacer lo siguiente?:

```
areaR :: a -> a -> a
areaR base altura = base * altura
```

Como *a* puede ser cualquier tipo, en particular podría ser un tipo que no podamos multiplicar, por ejemplo un *Char* o un *String*.

En el ejemplo de `append`, `fst` y `snd` no teníamos este problema pues no hacemos un uso específico del tipo, solo devolvemos los valores. En el caso de la función `áreaR` queremos multiplicar la base y la altura, situación que sólo tiene sentido en tipos numéricos, por lo que es incorrecto usar variables de tipo. Para solucionar problemas de este estilo utilizamos las clases de tipos.

Clases de tipos

Una clase es una *colección de tipos* que soportan ciertas operaciones sobrecargadas llamadas métodos. Si un tipo T es miembro de una clase C, decimos que T es instanica de C. Esto significa que ese tipo soporta e implementa el comportamiento que define la clase.

Para ejemplificar esta definición veremos las clases básicas de *Haskell*, su comportamiento, los métodos con los que cuentan y los tipos que pertenecen a cada clase:

1. **Eq - Tipos de igualdad:**
   - **Comportamiento:** Contiene tipos cuyos valores pueden ser comparados por igualdad o desigualdad.
   - **Métodos:**
     - (==) :: a −> a −> **Bool**
     - (/=) :: a −> a −> **Bool**
   - Todos los tipos básicos son instancia de esta clase, también las listas y las tuplas siempre que sus componentes o elementos sean instancia de esta clase.
2. **Ord - Tipos ordenados**
   - **Comportamiento:** Contiene tipos que son instancia de la clase Eq, pero además cuyos valores están totalmente ordenados (linealmente) y que puedan ser comparados.
   - **Métodos:**
     - (<) :: a −> a −> **Bool**
     - (>) :: a −> a −> **Bool**
     - (<=) :: a −> a −> **Bool**
     - (>=) :: a −> a −> **Bool**
     - (**min**) :: a −> a −> a
     - (**max**) :: a −> a −> a
   - Todos los tipos básicos son instancias de la clase Ord, también las listas y las tuplas siempre que sus componentes o elementos sean instancia de esta clase.
3. **Show:**

- **Comportamiento:** Contiene tipos cuyos valores pueden ser convertidos en una cadena de caracteres, por lo tanto se pueden mostrar en pantalla.

- **Método:**
  - **show** :: a $->$ **String**
- Todos los tipos básicos son instancia de esta clase, también las listas y las tuplas siempre que sus componentes o elementos sean instancia de esta clase.
  Es importante recalcar que los tipos función no pertenecen a la clase Show, por lo que si deseamos mostrar una función en pantalla tenemos que implementar este proceso manualmente.

4. **Read:**

- **Comportamiento:** Contiene tipos cuyos valores pueden ser convertidos de una cadena de caracteres a su tipo. Estos son los tipos que se pueden leer desde la consola.
- **Método:**
  - **read** :: **String** $->$ a
- Todos los tipos básicos son instancia de esta clase, también las listas y las tuplas siempre que sus componentes o elementos sean instancia de esta clase.

5. **Num - Tipos numéricos**

- **Comportamiento:** Contiene tipos que son instancia de la clase Eq y Show, pero además cuyos valores son númericos.
- **Métodos:**
  - (+) :: a $->$ a $->$ a
  - ($-$) :: a $->$ a $->$ a
  - ($*$) :: a $->$ a $->$ a
  - (**negate**) :: a $->$ a
    Devuelve el valor negado de un número, si es negativo se vuelve positivo y viceversa.
  - (**abs**) :: a $->$ a
    Devuelve el valor absoluto de un número.
  - (**signum**) :: a $->$ a
    Regresa 1 si el número es positivo, -1 si es negativo.
- Los tipos básicos Int, Integer y Float, son instancias de esta clase.

6. **Integral**

- **Comportamiento:** Contiene tipos que son instancia de la clase Num, pero cuyos valores son enteros.
- **Métodos:**
  - **div** :: a $->$ a $->$ a

- **mod** :: a −> a −> a

- Sólo los tipos Int e Integer son instancia de esta clase.

7. **Fractional**

   - **Comportamiento:** Contiene tipos que son instancias de la clase Num, pero cuyos valores no son enteros.
   - **Métodos:**
     - (/) :: a −> a −> a
     - (**recip**) :: a −> a
       Devuelve el recíproco de un número.
   - El tipo básico Float es instancia de esta clase.

Usando las clases de tipo, podemos solucionar nuestro error en la función que calcula el área de un rectángulo, pues restringiremos nuestra función a tipos númericos, es decir, tipos que pertenecen a la clase *Num*.

En general, para especificar que una variable de tipo, pertenece a una clase, debemos preceder los tipos de la función con: Class a =>, donde Class corresponde a una clase de las vimos anteriormente y *a* es la variable de tipo que pertenece a esta clase, seguido de las variables de tipo que espera la función, por ejemplo:

```
{- Firma de la funcion que espera un argumento de tipo "a"
   cuyo resultado es un elemento de tipo "a"
-}

Class a => a -> a
```

Dicho esto, la función que calcula el area de un rectángulo con variables de tipo y clases queda de la siguiente manera:

```
areaRec :: Num a => a -> a -> a
areaRec base altura = base * altura
```

Si queremos recibir tipos diferentes pero acotados por clases, hacemos lo siguiente:

```
divD :: (Fractional a, Integral b) => (a,b) -> (a,b)
divD (x,y) = ((x/2.3),(div y 2))
```

En este ejemplo estamos diciendo, que nuestro par recibe dos tipos diferentes, el primero que pertenezca a la clase *Fractional*, y el segundo a la clase *Integral*.

Tipos definidos por el usuario

Haskell nos permite crear nuestros tipos, hay tres maneras de hacerlo:

1. **Type:**
   La instrucción type nos permite renombrar los tipos básicos de *Haskell*, por ejemplo:

   type Base = **Float** - - Aquí estamos renombrando al tipo Float por Base
   type Nombre = **String** - - Aquí renombramos al tipo String como Nombre

2. **Newtype:**
   La instrucción newtype permite definir tipos usando un único constructor, el cual debe tener un único campo. Por ejemplo: **newtype** Lado = Altura **Int**

3. **Data:**
   La instrucción data, nos permite definir tipos nuevos utilizando múltiples constructores y recursión. El uso y funcionamiento de esta instrucción, se verá más adelante.

Casamiento de patrones (pattern matching)

Hasta ahora hemos mencionado este concepto un par de veces, pero, ¿qué es?. El casamiento de patrones es una construcción sintáctica de *Haskell*, la cual consiste en una especificación de pautas que deben seguir los datos[7], al momento de definir una función. El uso del casamiento de patrones tiene como ventaja simplificar la codificación ya que sólo escribimos la forma de lo que esperamos y podemos desglosar los componentes de cualquier tipo de dato primitivo de *Haskell* o tipos definidos por el usuario. .

Por ejemplo, cuando tenemos una tupla de aridad 5, *Haskell* no tiene una función que regrese cualquiera de sus elementos. Una manera fácil de definirla usando el casamiento de patrones es:

```
{- Funcion que dada una tupla de aridad 5 y un numero n
   (entre 1-5) regrese el elemento de la tupla indicado
   por el numero n.
-}

elementoTupla :: (a,a,a,a,a) -> Int -> a
elementoTupla (e1,e2,e3,e4,e5) 1 = e1
elementoTupla (e1,e2,e3,e4,e5) 2 = e2
elementoTupla (e1,e2,e3,e4,e5) 3 = e3
elementoTupla (e1,e2,e3,e4,e5) 4 = e4
elementoTupla (e1,e2,e3,e4,e5) 5 = e5
```

En este ejemplo podemos ver el uso del casamiento de patrones en dos tipos de datos, en el tipo tupla y en los enteros. El casamiento de patrones puede volverse tan complejo como sea el tipo.

El único incoveniente con su uso es que si cambiamos la estructura de un tipo para el cual ya hay funciones definidas usando casamiento de patrones, todas las funciones deben reescribirse adaptando el casamiento de patrones a la nueva estructura.

Funciones sobre listas

Listas

Como vimos anteriormente en *Haskell* tenemos el tipo lista y dadas las características del lenguaje, este tipo es la base para la mayoría de los programas que vamos a implementar.
De manera abstracta, una lista es una sucesión de datos de un mismo tipo, en donde importa el orden de los elementos.
Otra definición más formal es a partir de sus constructores:

- La lista vacía ([]) es una lista de tipo T.
- La lista que tiene una cabeza y una cola (x:xs), donde la *cabeza* (x) es un elemento de la lista, para ser más exactos el primero, y la *cola* (xs) es una sublista que contiene todos los elementos de la lista original menos el primer elemento.

El casamiento de patrones de *Haskell* va a reconocer los constructores de la lista [] y (x:xs) por lo que una función sobre listas generalmente se ve de la siguiente forma:

```
fun :: [a] -> T   – Firma de la función
fun [] = expresion_1     – Patrón para la lista vacía
fun (x:xs) = expresion_2  – Patrón para la lista no vacía
```

Funciones sobre listas

Haskell ya cuenta con funciones básicas sobre listas, veremos el comportamiento de algunas de ellas a continuación:

- **head**
  La funcion `head` regresa la cabeza de la lista.

```
Firma : head :: [a] -> a
Ejemplo :
Prelude> head [1,2,3,4]
1
```

- **tail**
  La función `tail` regresa la cola de la lista.

```
Firma : tail :: [a] -> [a]
Ejemplo :
Prelude> tail [1,2,3,4]
[2,3,4]
```

  Con estos ejemplos se ejemplifica lo que acabamos de mencionar, la cabeza de la lista tiene tipo a donde a es el tipo de los elementos de la lista y la cola tiene tipo $[a]$.

- **init**
  La función `init` regresa todos los elementos de una lista, menos el último.

```
Firma : init :: [a] -> [a]
Ejemplo :
Prelude> init [1,2,3,4]
[1,2,3]
```

- **length**
  La función `length` regresa la longitud (número de elementos) de la lista.

```
Firma : length :: [a] -> Int
Ejemplo :
Prelude> length [1,2,3,4]
4
```

- **(++)**
  La función `(++) o append` regresa la unión de dos listas, respetando el orden de ambas.

```
Firma : (++) :: [a] -> [a] -> [a]
Ejemplo :
Prelude> (++) [1,2,3] [4,5,6]
[1,2,3,4,5,6]
```

  Aunque podemos usar nuestra función de manera prefija lo más común es usar la función infija, de la siguiente manera:

```
Prelude> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
```

  Como podemos notar cuando queremos que una función comunmente infija se vuelva prefija, debemos ponerla entre paréntesis, como se muestra en el primer ejemplo.

- **reverse**
  La función `reverse` regresa la reversa de una lista.

```
Firma : reverse :: [a] -> [a]
Ejemplo :
Prelude> reverse [1,2,3,4]
[4,3,2,1]
```

- **drop**
  La función `drop` borra los primeros *n* elementos de la lista y regresa los elementos que sobran.

```
Firma : drop :: Int -> [a] -> [a]
Ejemplo :
Prelude> drop 3 [1,2,3,4,5,6]
[4,5,6]
```

- **take**
  La función `take` regresa una lista con los primeros *n* elementos de la lista que nos pasan como argumento.

```
Firma : take :: Int -> [a] -> [a]
Ejemplo :
Prelude> take 3 [1,2,3,4,5,6]
[1,2,3]
```

- **minimum**
  La función `minimum` regresa el menor elemento de una lista, para esto, los elementos de esta lista pertenecen a la clase **Ord**.

```
Firma : minimum :: Ord a => [a] -> a
Ejemplo :
Prelude> minimum [3,4,2,1,6,8,4,19,55]
1
```

- **maximum**
  La función `maximum` regresa el mayor elemento de una lista, para esto, los elementos de esta lista pertenecen a la clase **Ord**.

```
Firma : maximum :: Ord a=> [a] -> a
Ejemplo :
Prelude>  maximum [3,2,5,6,4,1]
6
```

- **sum**
  La función `sum` regresa la suma de los elementos de una lista, para esto, los elementos deben pertenecer a la clase **Num**.

```
Firma : sum :: Num a => [a] -> a
Ejemplo :
Prelude> sum [1,2,3,4]
10
```

Algunos ejemplos usando únicamente las funciones que acabamos de ver son:

- Definir la función `interior`, la cual recibe una lista (xs) y regresa los mismos elementos de la lista pero sin la cabeza ni el último elemento.

```
interior :: [a] -> [a]      – La firma no tiene restricciones en cuanto a tipos
interior [] = []              –Caso donde la lista que se recibe es la vacía
interior xs = tail (init xs) –Caso general

Ejemplo :

Prelude> interior [2,4,3,5,9,8]
[4,3,5,9]
```

- Definir la función `extremos`, la cual recibe un entero y una lista, y regresa la lista de los primeros y últimos *n* elementos de la lista original.

```
extremos :: Int -> [a] -> [a]– La firma no tiene restricciones de tipos
extremos n [] = [] –Caso donde la lista que se recibe es la vacía
extremos 0 l = l   –Caso donde no se piden extremos
extremos n xs = (take n xs) ++ drop ((length xs)-n) xs
                  –Caso general

Ejemplo :
Prelude> extremos 3 [2,3,5,2,1,4,6,2,5,6]
[2,3,5,2,5,6]
```

Listas por comprensión

Rangos

Los rangos, son una forma de crear una lista que contenga una secuencia aritmética de elementos enumerables. Por ejemplo para crear una lista de enteros del 1 al 100, hacemos lo siguiente:

[1..100]

El 1 indica el inicio de la lista, el 100 el final de la lista, y con los dos puntos (..) se indica en *Haskell* que la lista tiene todos los números intermedios entre 1 y 100.
Ejecutando esta instrucción en el interprete de *Haskell* tenemos:

```
Prelude> [1..100]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,
25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,
46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,
67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,
88,89,90,91,92,93,94,95,96,97,98,99,100]
```

También podemos especificar el número de pasos entre cada elemento, por ejemplo:

[2,4..20] [1,3..20] [1,4..20] [50,49..20]

Aquí se indica que entre el primer elemento de la serie y el segundo hay una especie de operación, y dicha operación se repetirá en el resto de la lista, por ejemplo la primera lista de los ejemplos pasados se vería como:

$$[l_0 = 2,\ l_1 = (l_0 + 2)\ ..\ l_9 = 20]$$

donde l_n indica la posición n de la lista.
Ejecutando las instrucciones en el interprete de *Haskell* obtenemos:

```
Prelude> [2,4..20]
[2,4,6,8,10,12,14,16,18,20]
```

```
Prelude> [1,3..20]
[1,3,5,7,9,11,13,15,17,19]
Prelude> [1,4..20]
[1,4,7,10,13,16,19]
Prelude> [50,49..20]
[50,49,48,47,46,45,44,43,42,41,40,39,38,37,36,35,34,33,32,
31,30,29,28,27,26,25,24,23,22,21,20]
```

También podemos ver en el ejemplo 2 y 3, que aunque la lista deba llegar a 20, al hacer la operación entre los elementos, el máximo número que podemos obtener sin pasarnos de 20 es 19, entonces este se convierte en el último elemento de la lista.
Un caso particular es entonces el ejemplo 4, pues creamos una lista en retroceso.

Notemos que únicamente especificamos un paso entre los elementos, con secuencias más complejas o que lleguen a ser ambiguas, no podemos usar este tipo de construcciones.

Listas infinitas

Si escribimos en el interprete de *Haskell* [1..] al no tener cota superior *Haskell* lo interpreta como un lista infinita y calculará todos los elementos de la misma, hasta donde los recursos de nuestra computadora se lo permitan.
Ahora si queremos únicamente los primeros 10 digitos de esa lista infinita, bastará con poner:

take 10 [1..]

Como mencionamos en el primer capítulo, *Haskell* cuenta con *evaluación perezosa*, esto quiere decir, que el intérprete o compilador de *Haskell* no va a crear toda la lista a menos que sea necesario, únicamente va a crear los elementos que utiliza, en el ejemplo anterior como sólo utiliza los primeros 10 elementos de la lista, son los únicos que creará para que take pueda calcularse, ejecutando esta instrucción en el interprete tenemos:

```
Prelude> take 10 [1..]
[1,2,3,4,5,6,7,8,9,10]
```

Listas por comprensión

Como ya vimos los rangos son una herramienta potente de *Haskell* pero limitada para sucesiones ambiguas o no tan directas. Otra manera de generar listas en *Haskell* es con las llamadas listas por comprensión, para entenderlas mejor, analizaremos su construcción desde las matemáticas.

En matemáticas, estamos acostumbrados a ver definiciones de conjuntos de este estilo:

$$\{x \in \mathbb{Z} \mid x = 2q+1, q \in \mathbb{Z}\}$$

$$\{2^x \mid x \in \mathbb{Z}, 0 \leq x \leq 10\}$$

El primero representa al subconjunto de los números impares pertenecientes a los enteros. Y el segundo representa al conjunto de potencias de 2, desde 2^0 hasta 2^{10}.

Haskell nos ofrece la manera de definir listas mediante este mismo mecanismo, usando como ya dijimos listas por comprensión, cuya estructura es la siguiente:

$$[f(x) \mid x \leftarrow \text{lista, predicados}]$$

donde :

- f es la función de salida, puede ser una función que se aplica a la variable como 2^x , o simplemente la variable x (función identidad, f(x) = x).
- x es una variable.
- La lista contendrá todos los elementos a los que se les aplicará la función de salida.
- x $\leftarrow$ lista, significa que los valores que tomará x serán todos los que estén en la lista.
- predicados son funciones que se evaluan a Bool, cuyo argumento es la variable. Sirven para hacer un filtro, a saber de los valores de la lista que cumplan con el predicado, podrán aplicarse a la función de salida. Son opcionales.

Analicemos la siguiente definición :

$$\{x \in \mathbb{Z}^+ \mid x = 2q+1, q \in \mathbb{Z}^+\}$$

- Función de salida: En este caso es la función identidad, pues regresamos a x.
- El conjunto con el que estamos trabajando es $\mathbb{Z}^+$
- Predicados: El predicado sería que x pueda representarse como “2q+1”, es decir, que x sea par.
- variable $\leftarrow$ lista: Todas las x que pertenezcan a $\mathbb{Z}^+$ $(x \in \mathbb{Z}^+)$

La traducción de este conjunto a su representación con listas por comprensión es:

$$[x \mid x <- [1..], \textbf{ even } x]$$

- Función de salida: Sigue siendo la función identidad como en el original.
- La lista que representa al conjunto $\mathbb{Z}^+$ es [1..].
- Predicados: La función `even` definida en el preludio de *Haskell*, regresa `True` si un número es par, `False` si no lo es.

Pese a ser una definición correcta, la ejecución de esta lista se ciclará por tener la lista infinita [1..], en estos caso, lo mejor es acotar nuestra lista.

Otro ejemplo es:

```
[x^2 | x <- [1..30]]
```

- Función de salida: En este caso la función de salida es x^2, el operador ∧ está definido en el preludio.
- La lista está representada por el rango [1..30]
- Predicados: No hay ningún predicado.

Si ejecutamos esta lista en el intérprete de *Haskell* obtenemos:

```
Prelude> [x^2 | x <- [1..30]]
[1,4,9,16,25,36,49,64,81,100,121,144,169,196,225,256,289,324,
361,400,441,484,529,576,625,676,729,784,841,900]
```

Veamos ahora un ejemplo con predicados:

```
[x | x <- [1..20], (mod x 3 == 0)]
```

- Función de salida: En este caso la función de salida es la función identidad
- Lista con los números del 1 al 20, representados con el rango [1..20]
- Predicados: (**mod** x 3 == 0): Todos los números de la lista que sean multiplos de 3.

Si ejecutamos esta lista en el intérprete de *Haskell* obtenemos:

```
Prelude> [x | x <- [1..20], (mod x 3 == 0)]
[3,6,9,12,15,18]
```

Listas por comprensión de varias variables

Hasta ahora hemos visto listas por comprensión con una variable, también tenemos la opción de crear listas con más de una variable, e incluso que dichas variables estén conectadas.
Un ejemplo clásico de listas por comprensión con dos variables es para obtener el conjunto cartesiano de dos listas.

```
[(x,y) | x <- [1..10], y <-[11..20], (even x)]
```

- Función de salida: En este caso la función de salida es la función identidad, pero formando con ambas variables un par.
- Tenemos dos listas, la lista a la que pertenece *x* es el rango [1..10] y a la que pertenece *y* es el rango [11..20].
- Predicados: Que x sea par, indicado con la función: `even x`

Si ejecutamos esta lista en el intérprete de *Haskell* obtenemos:

```
Prelude> [(x,y) | x <- [1..10], y <-[11..20], (even x)]
[(2,11),(2,12),(2,13),(2,14),(2,15),(2,16),(2,17),(2,18),
(2,19),(2,20),(4,11),(4,12),(4,13),(4,14),(4,15),(4,16),
(4,17),(4,18),(4,19),(4,20),(6,11),(6,12),(6,13),(6,14),
(6,15),(6,16),(6,17),(6,18),(6,19),(6,20),(8,11),(8,12),
(8,13),(8,14),(8,15),(8,16),(8,17),(8,18),(8,19),(8,20),
(10,11),(10,12),(10,13),(10,14),(10,15),(10,16),(10,17),
(10,18),(10,19),(10,20)]
```

La forma en la que las listas se recorren está indicada por el orden en el que aparecen los rangos, tomamos al primer elemento del primer rango y con este formamos los pares con todos los elementos del segundo rango y esto se repite con todos los elementos del primer rango, si cambiaramos el orden en el que escribimos los rangos obtendríamos la siguiente lista:

```
Prelude> [(x,y) | y <- [11..20], x <- [1..10], (even x)]
[(2,11),(4,11),(6,11),(8,11),(10,11),(2,12),(4,12),(6,12),
(8,12),(10,12),(2,13),(4,13),(6,13),(8,13),(10,13),(2,14),
(4,14),(6,14),(8,14),(10,14),(2,15),(4,15),(6,15),(8,15),
(10,15),(2,16),(4,16),(6,16),(8,16),(10,16),(2,17),(4,17),
(6,17),(8,17),(10,17),(2,18),(4,18),(6,18),(8,18),(10,18),
(2,19),(4,19),(6,19),(8,19),(10,19),(2,20),(4,20),(6,20),
(8,20),(10,20)]
```

Como podemos ver los elementos de la lista son los mismos, pero con diferente orden.

También podemos hacer que una variable dependa de otro rango aparte del suyo, por ejemplo[3]:

[(**toLower** y) | x <− [”HOLA”,”COMO”,”ESTAS”], y <− x]

- Función de salida: La función **toLower** que convierte un Char en mayúsculas a su equivalente en minúsculas.
- Tenemos dos listas, la lista a la que pertenece *x* es una lista de String (lista de Char) y la lista de *y* son todos los Char de la primera lista.
- Predicados: En este caso no hay predicados

Si ejecutamos esta lista en el intérprete de *Haskell* obtenemos:

```
Prelude> import Data.Char
Prelude Data.Char> [(toLower y) | x <- [”HOLA”,”COMO”,”ESTAS”],
                                  y <- x]
”holacomoestas”
```

Aquí tenemos dos variables, pero una de estas variables depende de la otra, se sigue ejecutando igual, para cada elemento del conjunto al que pertenece x, se recorren todos los elementos de la lista a la que pertenece y si en el ejemplo anterior intercambiamos el orden en el que están especificados los rangos obtenemos:

[3]Es necesario importar Data.Char, para poder usar la función toLower y toUpper

```
Prelude> import Data.Char

Prelude Data.Char> [(toLower y) | y <- x ,
                                  x <- ["HOLA","COMO","ESTAS"]]

<interactive>:6:21: error: Variable not in scope: x :: [Char]
```

Ocurre un error porque *y* necesita de *x* y *x* no se ha ejecutado, pues primero se hace lo correspondiente al rango de *y*.

Por último veremos que los rangos se pueden pasar como argumento, recibiremos un String y todos sus elementos los volveremos mayúsculas, la función y su ejecución quedan de la siguiente forma:

```
import Data.Char

mayusculas :: String -> String
mayusculas s = [toUpper x | x <- s]

--Ejecucion
*Ejemplo> mayusculas "hola"
"HOLA"
```

Otro ejemplo sería:

```
pares :: [Int] -> [Int]
pares l = [x | x <- l, (even x)]

--Ejecucion
*Ejemplo> pares [1..40]
[2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40]
```

Recursión

Definiciones recursivas

¿Qué es la recursión?

La recursión es, de manera muy general, un método de definición en donde usamos el concepto definido en la misma definición, es decir, la autoreferenciación.

Definición recursiva

Para construir una función recursiva, debemos incluir en su definición dos partes muy importantes:

1. **Conjunto de casos base**
   Son casos simples donde la definición se da directamente, es decir, sin usar la autorreferencia.
2. **Conjunto de reglas recursivas**
   Se define un nuevo elemento de la definición a partir de funciones constructoras que reciben como argumentos elementos más pequeños (anteriores) ya definidos.

Ejemplos:

1. Números naturales.
   - **Conjunto de casos base**
     0 es un número natural.
   - **Conjunto de reglas recursivas**
     Si n es un número natural, al aplicar la función constructora sucesor a n, denotado $Suc(n)$, es un número natural.
2. El conjunto S formado por las duplas (a,b) tal que a y b pertenecen a los naturales y $(a+b)$ es impar.
   - **Conjunto de casos base**
     (1,0) pertenece a S, (0,1) pertenece a S.
   - **Conjunto de reglas recursivas**
     Si (a,b) pertenece a S, entonces (a+1, b+1) pertenece a S.

Definición recursiva de listas

Uno de los tipos básicos de *Haskell* muy importante y bastante utilizado son las listas, su definición es recursiva y está dada por las siguientes reglas:

- **Conjunto de casos base**
  La lista vacía [] es una lista de elementos de A.

- **Conjunto de reglas recursivas**
  Si *xs* es una lista y *a* es un elemento de A entonces $cons(a, xs)$ es una lista de elementos de A.
  Donde *cons* es la función constructora que agrega un elemento al inicio de una lista.

Definiciones recursivas en Haskell

En *Haskell* hay dos maneras de definir tipos "nuevos", la primera es usando la palabra reservada `type` y lo que hace es cambiar el nombre de tipos ya existentes en *Haskell*, por ejemplo:

```
type Tupla = (a,b)
type Entero = Int
```

Pero, para hacer definiciones recursivas como vimos en la sección anterior, contamos con la palabra reservada `data`.
Con `data` creamos nuevos tipos, y en particular tipos recursivos, estos tipos recursivos los hacemos siguiendo las definiciones recursivas como las que ya vimos.
La definición recursiva de listas de enteros usando data es:

```
data Lista = Nil | Cons Int Lista
```

Como podemos ver, la definición es bastante similar a la primera que vimos, `Nil` es el constructor equivalente a la lista vacía, `Cons` es el constructor equivalente a la función cons. Lo único diferente es que fijamos el tipo de las listas a enteros, lo que no pasaba en la definición teórica, para solucionar este problema están los tipos parametrizados[4], entonces la definición de lista con tipos parametrizados es:

```
data Lista a = Nil | Cons a (Lista a)
```

Donde *a* representa cualquier tipo, con esta definición podemos tener listas de cualquier tipo, teniendo así una definición polimórfica.

Funciones recursivas

Cuando definimos tipos o estructuras recursivamente podemos definir funciones sobre los mismos utilizando el casamiento de patrones: cada cláusula de la definición del tipo introduce un patrón que equivale a un caso de la función.

Para definir una función recursiva es necesario seguir dos pasos primordiales:

[4] Similar a las variables de tipo, los tipos parametrizados nos permiten definir un tipo general que dependa de otro sin especificar cuál es en un principio.

1. **Definir el conjunto de casos base**.
   Casos simples donde no es necesario usar la autorreferencia.

2. **Definir el conjunto de reglas recursivas**.
   Se usa la autorreferencia y se busca que la llamada recursiva sea en un elemento más pequeño.

Usando la definición recursiva de lista que hicimos anteriormente, construimos algunas funciones sobre este tipo:

```
data Lista a = Nil | Cons a (Lista a) --Definicion de lista

--Funcion que regresa la longitud de una lista
long :: Lista a -> Int
long Nil = 0
long (Cons a xs) = 1 + (long xs)
```

Verifiquemos lo necesario para que esta definición sea válida.

1. **Definir el conjunto de casos base**.
   long Nil = 0
   Aquí no usamos la autoreferencia pues simplemente regresamos 0.

2. **Definir el conjunto de reglas recursivas**.
   long (Cons a xs) = 1 + (long xs)
   Aquí usamos la autoreferencia, llamando a long de xs, que claramente es un elemento más pequeño.

Recursión multiple

Ya vimos como funciona la recursión con una estructura, en *Haskell* podemos hacer recursión en más de una estructura al mismo tiempo, siguiendo las siguientes reglas:

1. **Definir el conjunto de casos base**.
   Casos simples donde no es necesario usar la autorreferencia, a diferencia de la recursión en una sola estructura, aquí debemos poner los casos base para todas las estructuras en las que estemos haciendo recursión.

2. **Definir el conjunto de reglas recursivas**.
   Se usa la autoreferencia, buscando que las llamadas recursivas sean en elementos más pequeños.

Como podemos ver, es igual a las reglas que usamos para hacer recursión en una sola estructura, solo que debemos aplicarlas para todas las estructuras en las que hacemos recursión.

A continuación definimos la función `take` que recibe un entero positivo n y una lista xs y regresa los primeros n elementos de la lista xs, por lo que debemos hacer recursión sobre ambos argumentos.

```
take :: Int -> [a] -> [a]
take 0 xs = []
take n [] = error "No hay suficientes elementos"
take n (x:xs) = x:(take (n-1) xs)
```

Verifiquemos la validez de esta definición:

1. **Definir el conjunto de casos base**.
   take 0 xs = []
   take n [] = **error** "No hay suficientes elementos"
   No usamos la autorreferencia simplemente regresamos la lista vacía y un error.

2. **Definir el conjunto de reglas recursivas**.
   take n (x:xs) = x:(**take** (n-1) xs))
   Usamos la autorreferencia, haciendo más pequeño al número al restarle 1, y a la lista, al hacer la llamada recursiva con la cola *xs*.

Hay que tener cuidado pues en *Haskell* se permite la recursión general (haciendo cualquier clase de autorreferencia), por ejemplo:

```
fun :: Int -> Int
fun n = fun n
```

Esta clase de recursión puede resultarnos útil, pero corremos el riesgo de que nuestra función nunca termine, entonces, para el desarrollo de las prácticas de este manual no se permitirán esta clase de definiciones.

Funciones de orden superior

Como vimos en el primer capítulo, un tipo primitivo de *Haskell* es el tipo función. Cuando nosotros definimos nuestras funciones hemos usado todos los tipos primitivos que tiene *Haskell*, pero ¿puede un tipo función ser tipo de otra función?

Una función de orden superior es aquella que puede tener como argumentos a otras funciones o incluso regresar funciones como resultado.

Hasta ahora la firma de nuestras funciones siempre ha sido de la forma :

$$\text{fun} :: T_1 \to T_2 \to ... \to T_n$$

Donde $T_1 \to T_2 \to ... \to T_{n-1}$ son los argumentos de la función y T_n es el tipo del valor que regresa. Si queremos ahora que nuestra función reciba como argumento otra función, lo haremos de la siguiente forma:

$$\text{fun} :: (R_1 \to R_2) \to T_3 \to ... \to T_n$$

En este ejemplo, la función está representada por $(R_1 \to R_2)$, en particular es otra función que toma como argumento a R_1 y devuelve R_2, así mismo si queremos que nuestra función regrese una función tendríamos algo de la forma:

$$\text{fun} :: T_1 \to T_2 \to ... \to (R_1 \to R_2)$$

Aquí decimos que la función regresa otra función denotada por $(R_1 \to R_2)$. Como vimos en los dos ejemplos anteriores, vamos a distinguir a las funciones dentro de otras funciones por medio de paréntesis, en los ejemplos sólo vimos funciones con dos tipos, pero pueden recibir más e incluso funciones dentro de las funciones.
Veamos un ejemplo: en nuestro capítulo de listas por comprensión decimos que la lista puede tener un predicado, dicho predicado hace un filtro a los elementos del rango, por ejemplo la función que dada una lista de enteros, nos devuelva los números pares de la misma, está definida de la siguiente manera:

```
pares :: [Int] -> [Int]
pares l = [x | x <- l, mod x 2 == 0]
```

Con este mismo razonamiento, ¿cómo haríamos una función general que nos permita cambiar el predicado, de tal forma que podamos con la misma función decidir por ejemplo si un entero es par, impar o divisible entre 3?. Para resolver esto convendría poder pasar una función como argumento, la cual funge como predicado. La nueva función se ve de la siguiente forma:

```
filtraLista :: (Int -> Bool) -> [Int] -> [Int]
filtraLista f l = [x | x <- l, f x]
```

Si quisieramos usar nuestra función para obtener todos los números divisibles entre 3, creamos primero nuestra función de filtro :

```
divT :: Int -> Bool
divT x = mod x 3 == 0
```

Y ahora usamos nuestra función filtraLista de la siguiente forma:

```
*OrdenSH> filtraLista divT [1..30]
[3,6,9,12,15,18,21,24,27,30]
```

Ahora definimos la misma función, usando recursión en vez de listas por comprensión:

```
filtra :: (Int -> Bool) -> [Int] -> [Int]
filtra f [] = []
filtra f (x:xs) = if f x then x:(filtra f xs)
                  else filtra f xs
```

Una función aún más general, es decir, que reciba cualquier tipo de lista y pueda filtrar los elementos de ella con una función es la función `filter`[5], cuya definición es :

```
filter :: (a -> Bool) -> [a] -> [a]
filter f [] = []
filter f (x:xs) = if f x then x:(filter f xs)
                  else filter f xs
```

Esta última es un clásico ejemplo de funciones de orden superior.
Observese que, en el ejemplo anterior, la función `divT` se convirtió en un argumento de entrada de la función `filtraLista`, pero esto no siempre es necesario, podemos pasar como argumentos funciones sin necesidad de definirlas previamente, usando el mecanismo de lambdas que explicamos a continuación.

Lambdas

Las lambdas en *Haskell* son funciones anónimas, comúnmente se utilizan para las funciones de orden superior y su sintaxis es la siguiente :

$$(\backslash x \ -> e)$$

Donde *x* es el argumento que espera la función y e es el valor de correspondencia o valor de la función. Un ejemplo concreto sería una función que realice lo mismo que `divT` :

[5]Esta función ya se encuentra definida en el preludio.

```
(\x -> mod x 3 == 0)
```

De esta forma ya no tenemos que crear más funciones de las necesarias, en particular funciones que sólo usamos una vez.

Si quisieramos ahora una función que realice lo mismo que nuestra función `filtraLista`, pero aplicándoles otra función a nuestros números filtrados, tendríamos algo así:

```
filtraL :: (Int -> Int) -> (Int -> Bool) -> [Int] -> [Int]
filtraL g f xs = [g x | x <- xs, f x]
```

Para probar nuestra función usamos lambdas de la siguiente forma :

```
*OrdenSH> filtraL (\x -> x ^ 2) (\x -> mod x 2 == 0) [1..30]
[4,16,36,64,100,144,196,256,324,400,484,576,676,784,900]
```

En el ejemplo anterior elevamos al cuadrado todos los números pares filtrados de nuestro rango [1..30].
Si quisieramos hacer nuestra función `filtraL` con tipos parametrizados, obtenemos:

```
filtraL2 :: (a -> b) -> (a -> Bool) -> [a] -> [b]
filtraL2 g f l = [g x | x <- l, f x]
```

Esta es una combinación entre `filter` y otra función de orden superior bastante conocida `map`, la cual dada una función f que va de A a B y una lista de tipo A, ℓ_A, le aplica a todos lo elementos de ℓ_A la función f, obteniendo así una nueva lista con elementos de tipo B, ℓ_B.

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = (f x): (map f xs)
```

Como siempre, al ser *Haskell* un lenguaje fuertemente tipado, debemos tener cuidado con los tipos que recibe la función principal, y los tipos de nuestras funciones argumento.

Definiciones locales

Veamos ahora las dos manera de emplear definiciones locales o bloques en *Haskell*.

Where

Así como las lambdas son una herramienta de *Haskell* para poder pasar funciones como argumento, `where` es una forma de hacer funciones anónimas[6] dentro de una función, veremos el uso de `where` haciendo la función pares, pero usando una función anónima.

```
--Funcion que regresa todos los pares de una lista de enteros.
paresW :: [Int] -> [Int]
paresW [] = []
paresW (x:xs) = if par x then x:(paresW xs)
```

[6]No se puede hacer uso de la función definida con `where` fuera de la función donde se definió.

```
                    else paresW xs
                    where par x = mod x 2 == 0
```

Las funciones definidas con `where` pueden ser tan sencillas como la anterior, o incluso ser una función recursiva que utiliza casamiento de patrones, esto se vería de la siguiente manera :

```
paresW' :: [Int] -> [Int]
paresW' [] = []
paresW' (x:xs) = if par x then x:(paresW' xs)
                 else paresW' xs
                 where
                     par 0 = True
                     par 1 = False
                     par n = par (n-2)
```

Al igual que con las lambdas, nos conviene el uso de `where` cuando usamos la función sólo una vez, pues al ser una definición local, las funciones definidas con `where` fuera de la función principal no existen.

Let

Al igual que con `where` podemos definir expresiones locales con `let`, usando la siguiente sintaxis:

let <definicion> **in** <expresion>

Las variables que definamos en la expresión después de `let` son accesibles únicamente en la expresión después de `in`.
En concreto `let` sirve para ligar variables en cualquier lugar, la diferencia con `where` es que las expresiones `let` son expresiones por si mismas mientras que las secciones `where` son simplemente construcciones sintácticas.[7]
Para algunas ocasiones el uso de `where` y `let` es prácticamente el mismo, por ejemplo para la función `paresW`, podemos hacer su función equivalente usando `let`.

```
--Funcion que regresa todos los pares de una lista de enteros.
paresL :: [Int] -> [Int]
paresL [] = []
paresL (x:xs) = let par = (mod x 2) == 0
                in if par then x:(paresL xs)
                   else paresL xs
```

Sin embargo, no podemos hacer una función equivalente a `paresW'` pues el uso de `where` en esta función no está ligando ninguna variable, por lo tanto, no puede sustituirse con `let`.

Funciones como resultado

Hasta ahora hemos visto las funciones de orden superior que reciben funciones como argumento, pero como dijimos en un inicio, también las funciones de orden superior son aquellas que

pueden regresar funciones.

Declaraciones locales definidas con `where` e incluso `let` resultan útiles para las funciones de orden superior que devuelven una función, pues podemos devolver las funciones definidas con ellas, por ejemplo, usando `where` definimos una función que dado un caracter regrese la operación correspondiente al mismo.

```
{-
 La funcion esta definida unicamente para suma, resta y
 multiplicacion.
-}

traductor :: Char -> (Int -> Int -> Int)
traductor '+' = suma
                where
                  suma n m = n + m
traductor '-' = resta
                where
                  resta n m = n - m
traductor '*' = multi
                where
                  multi n m = n * m
```

Lo complicado de este tipo de definiciones es que no es fácil visualizar sus resultados. Si intentaramos probar nuestra función traductor, obtendríamos lo siguiente :

```
*OrdenSH> traductor '+'

<interactive>:24:1:
    No instance for (Show (Int -> Int -> Int))
    (maybe you haven't applied enough arguments to a function?)
     arising from a use of print
    In a stmt of an interactive GHCi command: print it
```

Este error significa que *Haskell* no sabe como mostrar el tipo función, entonces a menos que definamos una instancia de **Show** para nuestras funciones, no es posible visualizar el resultado, entonces, la mejor forma de comprobar su funcionamiento es usándolas en otras funciones o usarlas como funciones parcialmente aplicadas.

Funciones parcialmente aplicadas

Otra manera de tener funciones que devuelvan funciones es utilizando funciones parcialmente aplicadas, es decir, llamar a una función con argumentos de menos.

En el ejemplo del traductor, regresamos una función dependiendo del caracter que recibe de argumento, si llamamos a la función con el argumento que necesita por ejemplo: traductor '+', obtendremos una función parcialmente aplicada, pues la función que regresa está esperando dos

argumentos y no los estamos pasando a la función.
Si a la función traductor le pasamos el argumento que espera y además los dos argumentos que espera la función que devuelve, obtenemos:

```
*OrdenSH> traductor '+' 5 8
13
```

En *Haskell* todas las funciones reciben un solo argumento, como es el caso de traductor, como mencionamos antes, para indicar funciones de múltiples argumentos tenemos la *currificación* y lo que hace en realidad es generar funciones parcialmente aplicadas hasta obtener el número de argumentos necesarios para ejecutar la función.

PARTE II

PRÁCTICAS

Práctica 1

Gramáticas

Objetivo

Implementar en Haskell *gramáticas formales* utilizando las definiciones de tipos `data`, `type` y *recursión*.

Preliminares

Una gramática formal es un mecanismo sencillo de especificación de reglas de construcción de palabras, llamadas *producciones* o *reglas de reescritura*, con las cuales se pueden generar expresiones de un lenguaje. [1].

Las reglas de reescritura son de la forma :

$$Simbolo ::= Cadena$$

Tenemos dos clases de símbolos, los *terminales* y los *no terminales*. Los terminales son aquellos que ya no pueden ser reescritos, es decir, no aparecen del lado izquierdo de una regla[1], mientras que los no terminales son los que pueden ser reescritos, en particular con ellos vamos a hacer las sustituciones de símbolos por cadenas.
Veamos como ejemplo una gramática que genera expresiones aritméticas en notación infija.

- E :: = var
- E ::= const
- E ::= $\triangleright$ E
- E ::= E $\diamond$ E
- E ::= (E)

[1] En otras gramáticas se permite el uso de terminales del lado izquierdo de la regla. No se considerarán dichas gramáticas para fines de este curso.

- var :: = **a** | **b** | ...
- const ::= **0** | **1** | **2** | **17** | **35** | ...
- $\triangleright$ ::= **+** | **-**
- $\diamond$ ::= + | - | $\times$ | $\div$

Implementación

La traducción de este tipo de reglas de reescritura en *Haskell* se hace con las definiciones de tipos `data` y `type`, las cuales nos brindan la forma de renombrar tipos (`type`) o crear nuestros propios tipos (`data`). Las gramáticas las podemos ver como tipos nuevos, por lo que estas herrmientas son de utilidad.

Para simplificar la traducción, utilizaremos únicamente números enteros, entonces, la traducción de la gramática de expresiones aritméticas a *Haskell* es la siguiente:

```
data Exp  = V Vars | Cons Int | S Signo Exp | O Oper Exp Exp |
            Par Exp deriving Show

data Vars = X Int deriving Show

data Signo = Pos | Neg deriving Show

data Oper = Suma | Resta | Mul | Div deriving Show
```

Como podemos ver es muy parecida a la gramática en la teoría, pero tenemos que hacer ciertos cambios en la adaptación para que cumpla con las reglas de *Haskell*. En la siguiente tabla enlistamos todas las reglas de la gramática del ejemplo y cómo fue su traducción a *Haskell*.

Gramática	*Haskell*	
E :: = var	**data** Exp = V Vars | ... **data** Vars = X **Int**	Como var es un símbolo no terminal, creamos otro data para representarlo, para facilitarnos la representación de variables de la forma $a, b, c...$, y que haya un número ilimitado de ellas, cambiamos la representación por variables de la forma $x_1, x_2, x_3...$
E ::= const	**data** Exp = Cons **Int** | ...	Como usamos únicamente enteros, definimos dentro de nuestro tipo Exp el constructor Cons, con el cual representaremos a nuestros números.
E ::= $\triangleright$ E	**data** Exp = S Sig Exp | ...	El símbolo $\triangleright$ también es un símbolo no terminal, por lo tanto, tenemos que crear otro data para representarlo, el cual únicamente tendrá los símbolos términales **Pos** y **Neg** que representan positivo y negativo respectivamente
E ::= E $\diamond$ E	**data** Exp = O Oper Exp Exp | ... **data** Oper = Suma | Resta | Mul | Div	Al igual que en el caso anterior, el símbolo $\diamond$ es un símbolo no terminal, entonces creamos un data para representarlo, este contara con los símbolos terminales Suma, Resta, Mul y Div, representando +, -, *, $\div$ respectivamente.
E ::= (E)	**data** Exp = Par Exp | ...	Aquí usamos el constructor **Par** para reprensentar que toda la expresión se encuentra entre paréntesis, no es necesario crear un constructor por cada paréntesis.

Los símbolos V, S, O, Cons y Par, son los constructores que necesita la definición de tipo `data`, por lo que no podemos omitirlos.
Algunos ejemplos de expresiones en nuestra gramática en *Haskell* son los siguientes:

```
-- (30 + 21)
ejem1 = Par $ O Suma (Cons 30) (Cons 21)

-- (X2 - 3)
ejem2 = Par $ O Resta (V (X 2)) (Cons 3)

-- (x1*(x2+34))
ejem3 = S Neg $ Par $ O Mul (V (X 1))
                        (Par (O Suma (V (X 2)) (Cons 34)))
```

Tal cual como está nuestra gramática no podemos visualizar el resultado en el intérprete pues *Haskell* no sabe como mostrarla, para lograrlo debemos agregar **deriving Show** al final de la declaración de cada data.

```
data Exp  = V Vars | Cons Int | S Signo Exp | O Oper Exp Exp |
            Par Exp deriving Show
```

```
data Vars = X Int deriving Show

data Signo = Pos | Neg deriving Show

data Oper = Suma | Resta | Mul | Div deriving Show
```

De esta forma al interpretar nuestros ejemplos tenemos:

```
*Gramaticas> ejem1
Par (O Suma (Cons 30) (Cons 21))
*Gramaticas> ejem2
Par (O Resta (V (X 2)) (Cons 3))
*Gramaticas> ejem3
S Neg (Par (O Mul (V (X 1)) (Par (O Suma (V (X 2)) (Cons 34)))))
```

Pero si quisieramos mostrar nuestra gramática de forma más estilizada, tendriamos que instanciar la clase **Show** directamente, por ejemplo:

```
-- Instancia de Show usando la función showEx
instance Show Exp where
      show x = showEx x

-- Función showEx para mostrar expresiones
showEx ::Exp -> String
showEx (V x) = showV x
showEx (Cons c) = show c
showEx (S Pos e) = showEx e
showEx (S Neg e) = "-" ++ showEx e
showEx (O Suma e1 e2) = (showEx e1) ++ "+" ++(showEx e2)
showEx (O Resta e1 e2) = (showEx e1) ++ "-" ++(showEx e2)
showEx (O Mul e1 e2) = (showEx e1) ++ "*" ++(showEx e2)
showEx (O Div e1 e2) = (showEx e1) ++ "/" ++(showEx e2)
showEx (Par e) = "(" ++ (showEx e) ++ ")"

-- Función auxiliar showV para mostrar las variables
showV :: Vars -> String
showV (X x) = "X" ++ (show x)
```

Cuando instanciamos la clase **Show**, nuestra gramática debe quedar como en el primer ejemplo, es decir, sin que diga **deriving Show**, pues de otro modo *Haskell* no sabe cual instancia de **Show** utilizar, si la que se incluye por defecto o la que acabamos de definir. Teniendo esto en cuenta si probamos nuestros ejemplos en el intérprete, obtenemos:

```
*Gramaticas> ejem1
(30+21)
*Gramaticas> ejem2
```

```
(X2-3)
*Gramaticas> ejem3
-(X1*(X2+34))
```

Gramática de paréntesis balanceados

Considera la siguiente gramática:

- B := ε | (R B
- R :=) | (R R

Cuya traducción a *Haskell* y un ejemplo son::

```
data B = Nil | LP R Par deriving Show
data R = RP | LP2 R R deriving Show


-- Ejemplo de la cadena (())() usando nuestros data para
-- generarla:
ejemP1 = LP (LP2 RP RP) (LP RP Nil)

-- Ejecución en el intérprete de nuestro ejemplo
*Main> ejemP1
LP (LP2 RP RP) (LP RP Nil)
```

A continuación veremos de ejemplo la función `contador`, la cual dada una cadena de paréntesis balanceados, cuenta el número de paréntesis que tiene dicha cadena.

```
-- Función contador
contador :: Par -> Int
contador Nil = 0
contador (LP r p) = 1 + (cuentaR r) + (contador p)

-- Función auxiliar cuentaR
cuentaR :: R -> Int
cuentaR RP = 1
cuentaR (LP2 r1 r2) = 1+ (cuentaR r1) + (cuentaR r2)

-- Ejecución con la cadena (())()
*Gramaticas> contador $ LP (LP2 RP RP) (LP RP Nil)
6
```

Ejercicios

1. Extiende la gramática de expresiones aritméticas, de tal forma que ahora acepte expresiones aritmético-lógicas. Agregando los operadores lógicos $\neg, \wedge, \vee$ y las constantes `True` y

`False`, para estas últimas no se pueden utilizar las constantes lógicas ya establecidas por *Haskell*.

2. Construye las siguientes cadenas a partir de la traducción de la gramática de paréntesis balanceados.
   - ()()()
   - (()())()
   - ()(())()

3. **showPar**

   Tipo : showPar :: Par −> **String**

   Especificación : Genera una cadena de paréntesis, a partir de la gramática de paréntesis balanceados.

   Ejemplo :

```
*Gramaticas> showPar $ LP (LP2 RP RP) (LP RP Nil)
"(())()"
```

4. Usando la función anterior, instancía la clase **Show** para la gramática de paréntesis balanceados.

5. Traduce las siguientes gramáticas a su equivalente en *Haskell* como se vio anteriormente:

   a)
   - S ::= [] | N : S | (S)
   - N ::= D | ND
   - D ::= 0 | 1 | ... | 9

   b)
   - S ::= A | O | (T
   - T ::=) | ST
   - O ::= + | - | * | ÷
   - A ::= 0 | 1 | ... | 9

   c)
   - S ::= FN | FL | Jk
   - F ::= ♣ | ♡ | ♠ | ♢
   - N ::= 2 | 3 | ... | 10
   - L ::= A | J | Q | K

6. Según tu traducción de las gramáticas anteriores, construye las siguientes cadenas.

   a)
   - ([1,9])
   - [0,1,2,3,4]

b)
- ((1+2)*3)
- (-(30+4))

c)
- ♡ 3
- ♠ A

Extras

1. Haz una función que dada una cadena generada por la gramática del ejercicio *6a*, regrese los números dentro de la cadena.
2. Realiza la función que regrese los operadores de las cadenas generadas por la gramática del ejercicio *6b*.
3. Elabora la función que dada una mano generada por la gramática del ejercicio *6c*, regrese un par cuyo primer elemento será la mano, y el segundo será la carta.
4. Crea la función que dada una mano, devuelva una lista de todas las cartas de dicha mano.
5. Haz el evaluador para las expresiones aritméticas representadas por la gramática de ejemplo en la sección implementación.

Cuestionario

1. ¿Por qué son necesarios los constructores en los tipos creados con `data`?
2. Da dos cadenas por cada gramática del ejercicio 6 además de las del ejercicio 7.
3. Muestra el proceso de derivavión de cada cadena del ejercicio anterior.
4. Dibuja el árbol de derivación de cada una de las cadenas generadas en el ejercicio anterior.

Práctica 2

Tablas de verdad

Objetivo

Implementar en Haskell *tablas de verdad de la lógica proposicional* y *tablas de verdad en el contexto de circuitos dígitales*, utilizando *recursión* y definiciones de tipos con `data`.

Preliminares

Para calcular la tabla de verdad de cualquier proposición E, es necesario considerar todos los estados posibles de los operandos de la expresión E. Cada operando puede estar en uno de los dos estados posibles, 1 verdadero o 0 falso. Cada renglón de la tabla corresponde a un estado particular de los operandos. Las tablas de verdad crecen tanto en columnas como en renglones, al volverse más compleja la fórmula en cuestión. El número de estados de una tabla estará dado por la función 2^n, donde n es el número de variables de la expresión.
Por ejemplo la tabla de la expresión $(\mathbf{P} \rightarrow \mathbf{Q}) \vee \mathbf{R}$ queda de la siguiente forma :

P Q R	(P → Q)	∨	R
0 0 0	1	**1**	0
0 0 1	1	**1**	1
0 1 0	1	**1**	0
0 1 1	1	**1**	1
1 0 0	0	**0**	0
1 0 1	0	**1**	1
1 1 0	1	**1**	0
1 1 1	1	**1**	1

El primer renglón de la tabla representa el nombre de las variables, así como la fórmula de la cual obtenemos la tabla de verdad. El resultado en la columna verde, es el de evaluar el estado del renglón en la fórmula, en el ejemplo está separada cada subfórmula para un mejor entendimiento, pero es equivalente a únicamente escribir:

P Q R	E
0 0 0	**1**
0 0 1	**1**
0 1 0	**1**
0 1 1	**1**
1 0 0	**0**
1 0 1	**1**
1 1 0	**1**
1 1 1	**1**

Donde E = (P $\rightarrow$ Q) $\vee$ R.

En lógica proposicional debemos de conocer la proposición para construir una tabla de verdad, en circuitos digitales podemos usar tablas de verdad para auxiliarnos a obtener la fórmula que resuelva cierto problema, podemos hacerlo de dos formas, utilizando los mintérminos (conjunciones de variables y variables negadas) o maxtérminos (disyunciones de variables y variables negadas), por ejemplo :

P Q	E	
0 0	1	$\leftarrow$ mintérmino
0 1	0	$\leftarrow$ maxtérmino
1 0	0	$\leftarrow$ maxtérmino
1 1	1	$\leftarrow$ mintérmino

A partir de esta tabla, si queremos obtener E utilizando *mintérminos*, escribimos las variables del renglón como una conjunción, dichas variables serán verdaderas o falsas dependiendo del estado, en el caso del ejemplo tenemos: $\bar{P}\bar{Q}$ y PQ, y las unimos con una disyunción, por lo tanto F = $\bar{P}\bar{Q} + PQ$. [1] Utilizando *maxtérminos*, escribimos igualmente las variables de cada renglón pero como disyunción y las unimos con conjunciones de la siguiente forma: $(\bar{P} + Q)(P + \bar{Q})$.

Implementación

Usaremos el siguiente tipo de dato para representar todos los estados posibles de la función:

```
type Renglon = ([Bool],Bool)
```

Donde [Bool] corresponderá al renglón de la tabla de verdad y Bool representará el estado en el que se encuentra dicho renglón.

```
type Tabla = [Renglon]
```

De esta forma la tabla estará compuesta por una lista de renglones. Como haremos nuestras tablas a partir de fórmulas proposicionales, utilizaremos los siguientes tipos de datos para representarlas:

[1] Recordemos que en circuitos digitales trabajamos con fórmulas en álgebra booleana equivalentes a fórmulas proposicionales utilizando únicamente and, or y negación, lo único que cambia es su representación.

```
data Prop = T | F | V NVar | And Prop Prop | Or Prop Prop |
            Not Prop | Imp Prop Prop | Eq Prop Prop
            deriving (Eq,Show)

data NVar = P Int deriving (Eq,Show)
```

Con `NVar` todas nuestras variables proposicionales estan representadas por P_n para facilitar la implementación.

A continuación veremos un ejemplo de una función `vars` la cual, cuenta las variables de una proposición.

```
-- Función vars
vars :: Prop -> Int
vars T = 0
vars F = 0
vars (V n) = 1
vars (And p q) = (vars p) + (vars q)
vars (Or p q)  =  (vars p) + (vars q)
vars (Not p) = vars p
vars (Imp p q) = (vars p) + (vars q)
vars (Eq p q) = (vars p) + (vars q)

-- Ejecución de la fórmula (p0 ∧ p1)
*Tverdad> vars $ And (V (P 0)) (V (P 1))
2
```

Ejercicios

1. **eval**
   Tipo : eval :: [Bool] -> Prop -> Bool
   Especificación : Evalua una proposición según un estado de sus variables.

   Ejemplos :

   ```
   *Tverdad> eval [True,False] $ And (V (P 0)) (V (P 1))
   False
   ```

2. **cons1**
   Tipo : cons1 :: Prop -> Tabla
   Especificación : Construye una tabla a partir de su proposición.

   Ejemplos :

   ```
   *Tverdad> cons1 $ And (V (P 0)) (V (P 1))
   [([False,False],False),
   ```

```
([False,True],False),
([True,False],False),
([True,True],True)]
```

Para este ejercicio, auxiliarse con las funciones 1 y 2 de los ejercicios.

3. **conv**
   Tipo : conv :: Int −> Int −> [Bool]
   Especificación : Convierte el primer entero a un número binario, representado por una lista de valores booleanos. El segundo entero nos sirve para determinar el número de bits que tendra dicho binario, por ejemplo si queremos representar al número 3 con 5 bits.

   Ejemplos :

```
*Tverdad> conv 3 5
[False,False,False,True,True]
```

4. Considera que los renglones de una tabla de verdad de n variables se pueden representar en forma decimal de tal manera que si tenemos dos variables la tabla tendrá cuatro renglones representados como:

0	00
1	01
2	10
3	11

Con base en esta información implementa la función `cons2`.

cons2
Tipo : cons2 :: Int −> (Int −> Bool) −> Tabla
Especificación : A partir del número de variables n y una función $f : Int \to Bool$, construye la tabla de verdad de las n variables, cuyo resultado de cada renglón es la evaluación de f en el número del renglón.

Ejemplos:

```
*Tverdad> cons2 3 (\x-> mod x 2 == 0)
[([False,False,False],True),([False,False,True],False),
([False,True,False],True),([False,True,True],False),
([True,False,False],True),([True,False,True],False),
([True,True,False],True),([True,True,True],False)]
```

Este ejercicio se puede ver como un caso particular de `cons1`, sólo que como explicamos en los preliminares, así como los renglones fungen de estado de las variables, también pueden verse como un número, en este caso, la fórmula es una función cuyo argumento son estos números.

Extras

1. Cambia la función `cons1`, para que ahora reciba una función de `[Bool]` a `Bool`.
2. Realiza una función que a partir de la tabla obtenida en `cons2`, regrese una fórmula proposicional hecha con mintérminos.
3. Elabora una función que a partir de la tabla obtenida en `cons2`, regrese una fórmula proposicional hecha con maxtérminos.
4. Define la instancia de **Show** para Tabla, de tal forma que tengan un aspecto más parecido a las tablas teóricas.

Cuestionario

1. ¿Por qué es necesario el segundo entero en la función `conv`?
2. ¿Puedes utilizar tablas de verdad en algún otro contexto de lógica?
3. ¿Qué es una forma normal?
4. ¿En que forma normal se encuentra la fórmula obtenida con mintérminos?
5. ¿En que forma normal se encuentra la fórmula obtenida con maxtérminos?

Práctica 3

Tableaux

Objetivo

Implementar *tableaux de la lógica proposicional* en Haskell, utilizando *recursión* y definicines de tipo `data`.

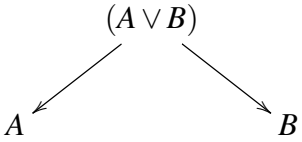
Preliminares

Un tableau corresponde a un árbol que dada una fórmula, busca una interpretación o estado que satisface dicha fórmula, este mecanismo permite determinar si una fórmula es tautología, contradicción o contingencia.[1]

Las fórmulas con las que se construye el tableau deben consistir únicamente de conjunciones y disyunciones de literales o constantes. La construcción del tableau se realiza siguiendo ciertas reglas, las cuales veremos acontinuación.

Reglas de construcción de tableaux

1. La fórmula para la que deseamos contruir el tableau aparece como raíz del árbol.
2. Si el esquema de la fórmula es una disyunción $(A \vee B)$, de la raíz del subárbol se abren dos ramas, una para la fórmula A y otra para la fórmula B.



   Las reglas para construir tableaux a partir de disyunciones las llamaremos reglas β.
3. Si el esquema de la fórmula es una conjunción $(A \wedge B)$ se pone a uno de los operandos como hijo del otro.

Las reglas para construir tableaux a partir de conjunciones las llamaremos reglas α.

4. También hay reglas para fórmulas que contienen doble negación, a las que llamaremos reglas σ, estas reglas se verán con mayor detalle más adelante.

5. Llamaremos expansión al proceso de creación o modificación de ramas a partir de una fórmula.

6. Al momento de expandir una fórmula esta no podrá expandirse nuevamente.

7. Cada vez que encontramos en una rama del árbol, una literal y su literal complementaria, podemos *cerrar* esa rama y ya no extenderla más.

8. Mientras que una rama no cierre, debemos seguir expandiendo las fórmulas que contiene hasta tener únicamente literales.

Implementación

Como mencionamos antes, un tableau es un árbol, cuyos nodos están etiquetados con una fórmula: el nodo raíz está etiquetado por la fórmula original, mientras que los nodos internos con subfórmulas obtenidas a partir de las reglas α, β, σ.
Para nuestra implementación, los nodos estarán etiquetados con listas de fórmulas, generadas mediante las reglas de expansión α, β, σ, de la manera descrita acontinuación:

- Reglas α

Teoría	Práctica
$\alpha_1 \wedge \alpha_2$ $\downarrow$ α_1 $\downarrow$ α_2	$[(\alpha_1 \wedge \alpha_2) : xs]$ $\downarrow$ $[\alpha_1 : \alpha_2 : xs]$
$\neg(\alpha_1 \vee \alpha_2)$ $\downarrow$ $\neg\alpha_1$ $\downarrow$ $\neg\alpha_2$	$[\neg(\alpha_1 \vee \alpha_2) : xs]$ $\downarrow$ $[\neg\alpha_1 : \neg\alpha_2 : xs]$
$\neg(\alpha_1 \rightarrow \alpha_2)$ $\downarrow$ α_1 $\downarrow$ $\neg\alpha_2$	$[\neg(\alpha_1 \rightarrow \alpha_2) : xs]$ $\downarrow$ $[\alpha_1 : \neg\alpha_2 : xs]$

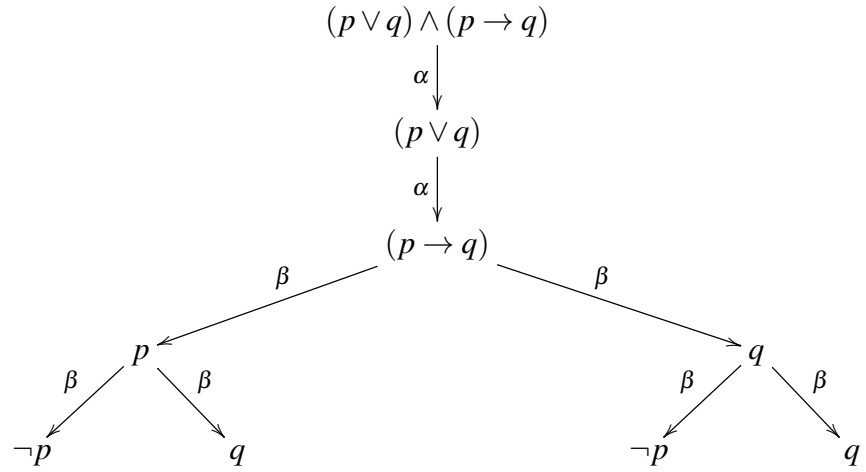
- Reglas β

Teoría	Práctica
$(\beta_1 \vee \beta_2)$ $\beta_1 \qquad \beta_2$	$[(\beta_1 \vee \beta_2) : xs]$ $[\beta_1 : xs] \qquad [\beta_2 : xs]$
$\neg(\beta_1 \wedge \beta_2)$ $\neg\beta_1 \qquad \neg\beta_2$	$[\neg(\beta_1 \wedge \beta_2) : xs]$ $[(\neg\beta_1) : xs] \qquad [(\neg\beta_2) : xs]$
$(\beta_1 \rightarrow \beta_2)$ $\neg\beta_1 \qquad \beta_2$	$[(\beta_1 \rightarrow \beta_2) : xs]$ $[\neg\beta_1 : xs] \qquad [\beta_2 : xs]$

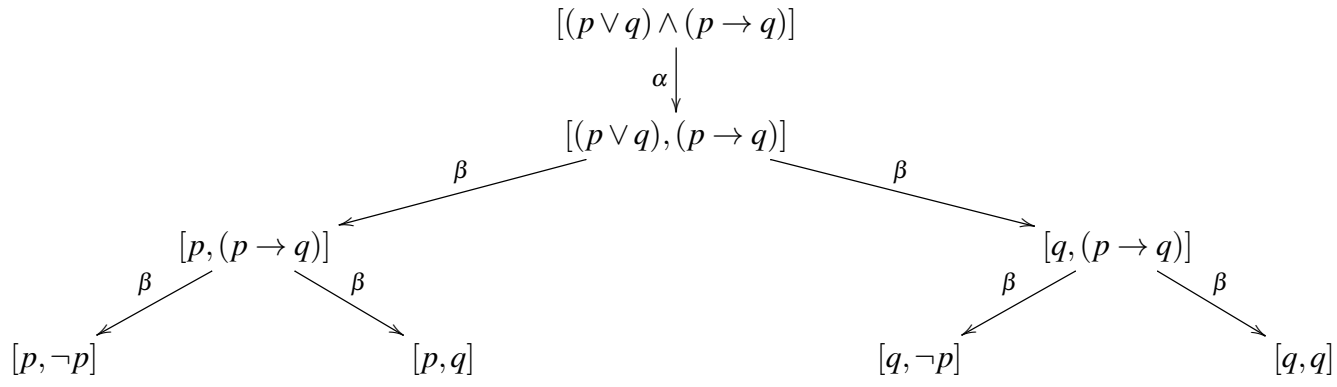
- Reglas σ

Teoría	Práctica
$\neg\neg\sigma$ $\downarrow$ σ	$[\neg\neg\sigma : xs]$ $\downarrow$ $[\sigma : xs]$
$\neg True$ $\downarrow$ $False$	$[\neg T : xs]$ $\downarrow$ $[F : xs]$
$\neg False$ $\downarrow$ $True$	$[\neg F : xs]$ $\downarrow$ $[T : xs]$

Es importante notar que si bien los esquemas anteriores muestran una expansión en listas a partir de la fórmula que se encuentra en su cabeza, no siempre es así, la fórmula a expandir puede ser un elemento intermedio de la lista. Esto quedará más claro con el siguiente ejemplo. En la teoría la fórmula: $(p \vee q) \wedge (p \rightarrow q)$ genera el siguiente Tableaux:



Este árbol corresponde en la implementación al siguiente árbol.



En el ejemplo podemos ver que hacer la expansión α representa eliminar de la lista la fórmula original y agregar las dos subfórmulas de manera independiente. La expansión β consiste en la creación de dos listas, ambas listas contendrán el resto de las fórmulas del tableau y la subfórmula correspondiente de acuerdo a la expansión β.

Los siguientes tipos de datos representarán a las fórmulas y los tableaux. Con base en ellos, realiza los ejercicios que se piden.

```
type Var = String

data LProp = T | F | VarP Var | Conj LProp LProp |
             Disy LProp LProp | Imp LProp LProp |
             Sii LProp LProp | Neg LProp deriving (Show,Eq)

data Tableaux = Hoja [LProp] | Alpha [LProp] Tableaux |
                Beta [LProp] Tableaux Tableaux deriving (Show,Eq)
```

Ejemplo, la función `literal` que nos dice si una proposición es una literal :

```
-- Función literal
literal :: LProp -> Bool
literal F = True
literal T = True
literal (VarP v) = True
literal (Neg (Neg v)) = False
literal (Neg v) = literal v
literal _ = False

--Ejecución de la función literal con ¬q
*Tableaux>literal $ Neg (VarP "q")
True
```

Ejercicios

1. **literales**
   Tipo : literales :: [LProp] -> **Bool**
   Especificación : Nos dice si en una lista de fórmulas, todas son literales.

   Ejemplos :

```
--literales [s,¬a] ⇒ True
*Tableaux>literales [VarP "s", Neg (VarP "q")]
True

--literales [p∨q] ⇒ True
*Tableaux> literales [Disy (VarP "p") (VarP "q")]
```

```
False
```

2. **nextF**
   Tipo : nextF :: [LProp] −> Prop
   Especificación : Regresa la primera fórmula que no es literal, de una lista de fórmulas.

   Ejemplos :

```
--nextF [p∧q,¬r] ⇒ (p∧q)
*Tableaux> nextF [(Conj (VarP "p") (VarP "q")),
                  Neg (VarP "r")]
Conj (VarP "p") (VarP "q")
```

3. **alpha**
   Tipo : alpha :: LProp −> **Bool**
   Especificación : Nos dice si una fórmula f es una fórmula alpha

   Ejemplos :

```
--alpha p∧q ⇒ True
*Tableaux> alpha $ Conj (VarP "p") (VarP "q")
True
```

4. **beta**
   Tipo : beta :: LProp −> **Bool**
   Especificación : Nos dice si una fórmula f es una fórmula beta

   Ejemplos :

```
--beta p∨q ⇒ True
*Tableaux> beta $ Disy (VarP "p") (VarP "q")
True
```

5. **sigma**
   Tipo : sigma :: LProp −> **Bool**
   Especificación : Nos dice si una fórmula f es una fórmula sigma

   Ejemplos :

```
--sigma ¬(¬(p∧q)) ⇒ True
*Tableaux> sigma $ Neg (Neg (Conj (VarP "p") (VarP "q")))
True
```

6. **expAlpha**
   Tipo : expAlpha :: [LProp] −> LProp −> [LProp]
   Especificación : Dada una lista de fórmulas ℓ y una fórmula f, realiza la expansión alpha de f dentro la lista ℓ.

   Ejemplos :

```
--expAlpha [(p∧q),¬r] (p∧q) ⇒ [p,q,¬r]
*Tableaux>expAlpha [(Conj (VarP "p") (VarP "q")),
                    Neg (VarP "r")]
                   (Conj (VarP "p") (VarP "q"))

[(VarP "p"), (VarP "q"), Neg (VarP "r")]
```

7. **expBeta**

   Tipo : expBeta :: [LProp] -> LProp -> ([LProp],[LProp])
   Especificación : Dada una lista de fórmulas ℓ y una fórmula f, realiza la expansión beta de f sobre la lista ℓ.

   Ejemplos :

```
--expBeta [s,(p∨q),r] (p∨q) ⇒ ([p,s,r],[q,s,r])
*Tableaux> expBeta [VarP "s", (Disy (VarP "p") (VarP "q")),
                    VarP "r" ] (Disy (VarP "p") (VarP "q"))

([VarP "p",VarP "s",VarP "r"],[VarP "q",VarP "s",VarP "r"])
```

8. **expSigma**

   Tipo : expSigma :: [LProp] -> LProp -> [LProp]
   Especificación : Dada una lista de fórmulas ℓ y una fórmula f, realiza la expansión sigma de f sobre la lista ℓ.

   Ejemplos :

```
--expSigma [¬(¬(p∧q))] (¬(¬(p∧q))) ⇒ (p∧q)
*Tableaux> expSigma [Neg (Neg (Conj (VarP "p") (VarP "q")))]
                    Neg (Neg (Conj (VarP "p") (VarP "q")))

[(Conj (VarP "p") (VarP "q"))]
```

9. **consTableaux**

   Tipo : consTableaux :: LProp -> Tableaux
   Especificación : Construye el tableau a partir de una fórmula.

   Ejemplos :

```
{--consTableaux (p∨((p∧¬q)→r)) ⇒  Beta [p∨((p∧q)→r)]
                                       (Hoja[p])
                                       (Beta [p∧q→r]
                                             (Beta [¬(p∧¬q)]
                                                   (Hoja [¬p])
                                                   (Hoja [q]))
                                             (Hoja[r]))
--}
```

```
*Tableaux> consTableaux $ Disy (VarP "p")
                          (Imp (Conj (VarP "p")
                               (Neg (VarP "q"))) (VarP "r"))

Beta [Disy (VarP "p")
           (Imp (Conj (VarP "p") (Neg (VarP "q")))
                (VarP "r"))]
     (Hoja [VarP "p"])
     (Beta [Imp (Conj (VarP "p") (Neg (VarP "q")))
            (VarP "r")]
           (Beta [Neg (Conj (VarP "p") (Neg (VarP "q")))]
                 (Hoja [Neg (VarP "p")])
                 (Hoja [VarP "q"]))
           (Hoja [VarP "r"]))
```

10. **cierra**

Tipo : cierra :: Tableaux −> **Bool**

Especificación : Verifica si todas las ramas del tableaux son cerradas.

Ejemplos :

$$\{--cierra\ \$\ Alpha\ [\neg(p \vee ((p \wedge \neg q) \to r))]\ (Alpha\ [\neg p, \neg((p \wedge \neg q) \to r)]\ (Alpha\ [p \wedge \neg q, \neg r, \neg p]\ (Hoja\ [p, \neg q, \neg r, \neg p]))) \Rightarrow True\ --\}$$

```
*Tableaux> cierra $ Alpha
                    [Neg (Disy (VarP "p")
                               (Imp (Conj (VarP "p")
                                          (Neg (VarP "q")))
                                    (VarP "r")))]
                    (Alpha [Neg (VarP "p"),
                            Neg (Imp (Conj (VarP "p")
                                           (Neg (VarP "q")))
                                     (VarP "r"))]
                           (Alpha [Conj (VarP "p")
                                        (Neg (VarP "q")),
                                   Neg (VarP "r"),
                                   Neg (VarP "p")]
                                  (Hoja [VarP "p",
                                         Neg (VarP "q"),
                                         Neg (VarP "r"),
                                         Neg (VarP "p")])))

True
```

11. **tautologia**
    Tipo : tautologia :: LProp −> **Bool**
    Especificación : Usando tableaux, verificamos si una fórmula es tautología.

    Ejemplos :

```
-- tautologia $ p∨((p∧¬q)→r)⇒True
*Tableaux> tautologia $ Disy (VarP "p")
                             (Imp (Conj (VarP "p")
                                        (Neg (VarP "q")))
                                  (VarP "r"))

True
```

Extras

1. Define la función que verifique que una fórmula es contradicción.
2. Elabora la función que verifique que una fórmula sea contingencia.
3. Si la fórmula original se encuentra en Forma Normal Negativa (formula sin implicaciones, equivalencias y dobles negaciones), varias reglas α, β, σ son innecesarias. Realiza la función FNN, la cual transforma la fórmula original en su equivalente en Forma Normal Negativa.
4. Otra manera de implementar Tableaux, es realizando las expansiones α, β, σ sobre la lista original, sin hacer uso del data Tableaux, realiza esta implementación.

Cuestionario

1. ¿Cuál es la diferencia entre las expansiones comunes en diferentes ramas entre teoría y práctica?
2. ¿Qué es necesario añadir para poder probar consecuencia lógica?

Práctica 4

Mapas de Karnaugh

Objetivo

Implementar el proceso de reducción de circuitos con *mapas de Karnaugh de dos variables* utilizando *recursión* y definiciones de tipo `data`.

Preliminares

En circuitos digitales, siempre es preferible trabajar con circuitos pequeños; podemos hacer una reducción de estos con métodos algebraicos, pero con ellos no podemos garantizar que lleguemos a una buena solución.
Los mapas de Karnaugh son un método gráfico para minimizar circuitos digitales. Como cada circuito tiene una función booleana que lo represente, para minimizar circuitos con mapas de Karnaugh, se trabaja con funciones booleanas las cuales deben encontrarse en *forma normal disyuntiva*.
El método consiste en los siguientes pasos:

1. Dependiendo del número de variables de la fórmula, se dibuja una cuadrícula de 2^n celdas, donde n es el número de variables.

2. Cada celda del mapa representa a un estado para evaluar la fórmula o *mintérmino*. La regla para armar los mapas, es que el cambio entre dos celdas consecutivas, ya sea horizontales o verticales, no puede ser más que de una variable.

3. Una vez hecha la cuadrícula, se marcan con el número uno los minterminos presentes en la fórmula a reducir.

4. Se deben encontrar bloques de unos de tamaño 2^i, donde $i \leq n$. Los bloques pueden ser horizontales o verticales, no diagonales. Se tratará de crear el grupo de unos más grande posible y los grupos pueden intersectarse.

5. Con los grupos hechos, procedemos a obtener la fórmula minimizada, para esto debemos observar las variables que permanecen constantes en los grupos, si dentro de un grupo

aparece una variable y su negación, se eliminará esta variable. Por cada grupo obtendremos un nuevo mintérmino. La fórmula reducida será la suma de todos los minterminos encontrados.

Nota: Si tenemos una fórmula de este estilo : $\bar{y} + xy + \bar{x}y$ donde el primer mintérmino sólo tiene a la variable $\bar{y}$, se debe multiplicar dicho mintérmino por el neutro multiplicativo $(x + \bar{x})$. Utilizamos a x en este caso por ser la variable que falta en el mintérmino, pero puede variar. Haciendo los cálculos correspondientes podemos continuar con el procedimiento antes descrito.

Implementación

Utilizaremos los siguientes tipos de datos para representar a las variables, las literales y las fórmulas admisibles para los mapas de Karnaugh:

```
data Var = X | Y    deriving (Show,Eq)
data Lit = Neg Var | V Var deriving (Show,Eq)
data FBool = L Lit | Prod FBool FBool  |
             Sum FBool FBool deriving (Show,Eq)
```

Con la finalidad de hacer más simple el proceso de los mapas, supondremos que nuestra fórmula se encuentra en forma normal disyuntiva, lo cual se refleja en la definición del tipo FBool. Cada conjunción de la fórmula será un mintermino dento del mapa y estos mintérminos estarán fijos de la siguiente manera:

11	10
xy	$x\bar{y}$
01	00
$\bar{x}$y	$\bar{x}\,\bar{y}$

Este mapa estará representado por una lista de tipo Bool, donde cada posición es un mintérmino. Si alguna posición es True, significa que el mintérmino que representa aparece en la fórmula, False en otro caso. Por lo tanto, el tipo para mapas de Karnaugh es:

```
type KMap = [Bool]
```

Y la posición de los mintérminos con respecto a la lista es:

Posición	Mintérmino
0	xy
1	$x\bar{y}$
2	$\bar{x}$y
3	$\bar{x}\,\bar{y}$

Por ejemplo, la fórmula $\bar{x}y + xy$ correspondería al mapa [**True**,**False**,**True**,**False**].

Ejercicios [1]

1. Transfoma las siguientes fórmulas a su representación con el tipo FBool.

   a) $\bar{x}y + xy$

   b) $\bar{x}y + x\bar{y}$

   c) $x\bar{y} + \bar{x}y + \bar{x}\,\bar{y}$

2. **consMap**

   Tipo : consMap:: FBool −> KMap

   Especificación : Esta función construye el mapa de Karnaugh, a partir de una fórmula. Cada posición será True o False, dependiendo de si el mintérmino está en la fórmula.

   Ejemplos :

```
--Donde a, b y c corresponden a las respuestas del ejercicio
-- anterior.
*Mapas> consMap a
[True,False,True,False]
*Mapas> consMap b
[False,True,True,False]
*Mapas> consMap c
[False,True,True,True]
```

3. **bloques**

   Tipo : bloques :: KMap −> [KMap]

   Especificación : Función que forma los bloques del mapa, estos bloques estarán representados por las posiciones del mapa que pueden formar el bloque. Para facilitar el ejercicio, realiza la pregunta 3 del cuestionario.

   Ejemplos :

```
--Donde a, b y c corresponden a las respuestas del
--ejercicio 1
*Mapas> bloques $ consMap a
[[0,2]]
*Mapas> bloques $ consMap b
[[2],[1]]
*Mapas> bloques $ consMap c
[[2,3],[1,3]]
```

4. **fMin**

   Tipo : fMin :: FBool −> FBool

   Especificación : Función que recibe una fórmula y la regresa minimizada utilizando nuestros mapas de Karnaugh.

[1] En todas las funciones se suponen fórmulas de únicamente dos variables.

Ejemplos :

```
--Donde a y c corresponden a las respuestas del
--ejercicio 1
*Mapas> fMin a
L (V Y)
*Mapas> fMin c
Sum (L (Neg X)) (L (Neg Y))
```

Extras

1. Haz una implementación que calcule los bloques para un mapa de 3 variables.
2. Realiza una implementación que obtenga la fórmula minimizada para un mapa de 3 variables.
3. Construye una implementación que calcule los bloques para un mapa de 4 variables.
4. Elabora una implementación que obtenga la fórmula minimizada para un mapa de 4 variables.
5. Define un tipo de datos para fórmulas en lógica proposicional, a partir de ella realiza la función que transforme cualquier fórmula a una en forma normal disyuntiva y otra a forma normal disyuntiva completa (ejercicio 6 del cuestionario).

Cuestionario

1. Si en la fórmula a reducir están todos los minterminos posibles. ¿A qué se reduce?, ¿Por qué?
2. ¿Cuántos bloques podemos obtener de un mapa de cuatro variables?
3. Dibuja con colores diferentes todos los posibles bloques de un mapa de 2 variables.
4. ¿Cuál es el principal problema en el proceso de mecanización del método de mapas de Karnaugh?
5. Investiga como funciona el método de mapas de Karnaugh para cinco variables.
6. Investiga cuál es la forma normal disyuntiva completa.

Práctica 5

Listas de longitud par

Objetivo

Implementar una modificación de la *estructura lista* utilizando *recursión*.

Preliminares

Las listas de longitud par, tienen las mismas características que las listas que ya hemos revisado en el curso, la única peculiaridad como su nombre lo indica, es que la cantidad de elementos dentro de la lista debe ser par, por lo tanto, la lista vacía se considera un elemento de esta estructura al ser cero un número par.
Para cumplir con este requisito, se debe cuidar que todas las operaciones sobre estas listas preserven el invariante de que la longitud sea par. Por ejemplo, no es posible definir una función que agregue un único elemento a la lista.

Implementación

Como mencionamos anteriormente, podemos añadir elementos a la lista, pero para conservar su propiedad de longitud par, siempre debemos añadir 2 elementos.
Para facilitar esta acción veremos esos elementos como una unidad, añadiendo un par (a,b), utilizando los pares que *Haskell* nos ofrece.
Cabe resaltar, que los elementos podrán buscarse de manera individual, es decir, para la implementación utilizaremos los pares, pero conceptualmente se pensará en una lista de elementos consecutivos. De esta forma la lista <1,2,3,4,5,6> estará dada por [(1,2),(3,4),(5,6)] .

Otro detalle que debemos tomar en cuenta, es que a diferencia de las listas comunes de *Haskell*, los índices de estas listas empiezan en 1, por motivos de conveniencia para la implementación.

De tal forma que utilizaremos el siguiente tipo de datos:

```
type EList a = [(a,a)]
```

Ejemplo, la función `longP` que calcula la longitud de una EList.

```
--Función longP
longP :: EList a -> Int
longP [] = 0
longP (x:xs) = 2 + longP xs

--Ejecución de la función longP con [(1,2),(3,4),(5,6)]
*EList> longP [(1,2),(3,4),(5,6)]
6
```

Ejercicios

1. **elemP**
   Tipo : elemP :: **Eq** a => a -> EList a -> **Bool**
   Especificación: Decidir si un elemento pertenece a una EList.

   Ejemplos :

```
*EList> elemP 3 [(1,2),(3,4),(5,6)]
True

*EList> elemP 9 [(1,2),(3,4),(5,6)]
False
```

2. **consP**
   Tipo : consP :: (a,a) -> EList a -> EList a
   Especificación: Agregar dos elementos al principio de la lista.

   Ejemplos :

```
*EList> consP (1,2) [(7,8),(9,10),(11,12)]
[(1,2),(7,8),(9,10),(11,12)]
*EList> consP (232,23) []
[(232,23)]
```

3. **appendP**
   Tipo : appendP :: EList a -> EList a -> EList a
   Especificación: Realiza la concatenación de dos listas EList.

   Ejemplos :

```
*EList> appendP [(1,2),(3,4),(5,6)] [(7,8),(9,10),(11,12)]
[(1,2),(3,4),(5,6),(7,8),(9,10),(11,12)]
```

4. **snocP**
   Tipo : snocP :: (a,a) −> EList a −> EList a
   Especificación: Agrega dos elementos al final de la lista.

   Ejemplos :

```
*EList> snocP (2,1) [(6,5),(4,3)]
[(6,5),(4,3),(2,1)]
*EList> snocP (12,754) [(1,2),(7,8),(9,10),(11,12)]
[(1,2),(7,8),(9,10),(11,12),(12,754)]
```

5. **atP**
   Tipo : atP :: EList a −> **Int** −> a
   Especificación: Regresa el n-ésimo elemento de una lista EList.

   Ejemplos :

```
*EList> atP [(7,8),(9,10),(11,12)] 3
9
```

6. **updateP**
   Tipo : updateP :: EList a −> **Int** −> a −> EList a
   Especificación: Cambia el n-ésimo elemento de una EList, por el elemento dado.

   Ejemplos :

```
*EList> updateP [(7,8),(9,10),(11,12)] 4 6
[(7,8),(9,6),(11,12)]

*EList> updateP [('h','o'),('l','a')] 3 'y'
[('h','o'),('y','a')]
```

7. **aplanaP**
   Tipo : aplanaP :: EList a −> [a]
   Especificación: Regresa una lista EList aplanada, es decir, todos los elementos de una EList, separados en una lista común.

   Ejemplos :

```
*EList> aplanaP [(1,2),(3,4),(5,6)]
[1,2,3,4,5,6]
```

8. **toEL**
   Tipo : toEl :: [a] −> EList a
   Especificación: Convierte una lista común en una EList, si la lista común es de longitud impar, no se agregará el último elemento.

   Ejemplos :

```
*EList> toEL [1,2,3,4,5]
[(1,2),(3,4)]

*EList> toEL "hola"
[('h','o'),('l','a')]
```

9. **dropP**

   Tipo : dropP :: **Int** -> EList a -> EList a
   Especificación: Borra los n primeros elementos de EList, donde n es par.

   Ejemplos :

```
*EList> dropP 2 [('h','o'),('l','a')]
[('l','a')]

*Main> dropP 3 (toEL "hola")
*** Exception: No es un numero par
```

10. **dropN**

    Tipo : dropN :: **Int** -> EList a -> [a]
    Especificación: Borra los n primeros elementos de EList, n puede
    ser par o impar, pues se regresará una lista común.

    Ejemplos :

```
*EList> dropN 3 [(1,2),(7,8),(9,10),(11,12),(12,754)]
[8,9,10,11,12,12,754]
```

11. **takeP**

    Tipo : takeP :: **Int** -> EList a -> EList a
    Especificación: Toma los n primeros elementos de una EList, n debe ser par.

    Ejemplos :

```
*EList> takeP 2 [(1,2),(3,4)]
[(1,2)]
```

12. **takeN**

    Tipo : takeN :: **Int** -> EList a -> [a]
    Especificación: Toma los n primeros elementos de una EList, n puede
    ser par o impar, pues los elementos se regresarán en una lista común.

    Ejemplos :

```
*EList> takeN 3 [(1,2),(3,4)]
[1,2,3]
```

Extras

1. Elabora la función que haga la reversa de una lista de longitud par.
2. Realiza la función que elimine los dos elementos medios de la lista. Debes considerar, que en algunos casos, los elementos medios pueden estar agrupados en diferentes pares, en cuyo caso deberas reconstruir la lista.
3. Construye la función que elimine cualesquiera dos elementos de la lista, se debe verificar que existan, y si están agrupados en pares diferentes, se deberá reconstruir la lista.
4. Haz la función `map` para esta estructura.
5. Define la función que borre los elementos al extremo de una lista.
6. Realiza la función que invierta todos los pares de la lista.

Cuestionario

1. Da otra idea para la implementación de listas de longitud par, sin utilizar pares.
2. Enlista tres propiedades ecuacionales que cumplan las operaciones de estas listas.
3. Demuestra alguna de las tres propiedades que enlistaste en el ejercicio anterior.

Práctica 6

Multiconjuntos

Objetivo

Implementar la estructura de datos *multiconjuntos* [1] utilizando las funciones básicas sobre *listas* y *tuplas* además *recursión*.

Preliminares

En matemáticas, cuando hablamos de conjuntos, nos referimos a una colección de elementos sin repeticiones. A diferencia del conjunto, un *multiconjuto* o *bolsa A* puede tener múltiples ocurrencias del mismo elemento.
Existen varias maneras de representar un multiconjunto, por ejemplo como una función de valor numérico, dicha función tendrá como dominio algunos elementos de A y el contradominio usualmente es el conjunto de los números naturales, de tal forma que al evaluar un elemento de A en la función, obtendremos el número de ocurrencias de dicho elemento en el multiconjunto.
Otra manera de representar los multiconjuntos es por medio de un conjunto de pares $\langle a,n\rangle$ donde a es un elemento de A y n denota el número de veces que aparece a en el multiconjunto. A este número lo denotaremos como *multiplicidad*. Debemos tener cuidado con esta definición pues el par $\langle a,0\rangle$ no está permitido, pues significa que el elemento a esta cero veces en el multiconjunto, es decir, no está.
Las operaciones básicas que conocemos sobre conjuntos, se mantienen en los multiconjuntos, adecuando su definición a la representación que le demos al multiconjunto.

A continuación enunciamos algunas operaciones de multiconjuntos [2].

- **Igualdad**
  Sean A y B multiconjuntos, $A = B$, si y solo si $m_A(x) = m_B(x), \forall x \in A, x \in B$.
- **Submulticonjunto**
  Sean A y B multiconjuntos, A es submulticonjunto de B, si y solo $m_A(x) \leq m_B(x), \forall x \in A,\ \ x \in B$.

[1] Se sufiere consultar el punto [6] de la bibliografía.
[2] Suponemos la operación $m_T(x)$, la cual representa la multiplicidad de x en el conjunto T.

- **Unión** Sean A y B multiconjuntos, en la unión de A y B (denotada $A \cup B$), para toda $x \in A \cup B$, $m_{A \cup B}(x) = m_A(x) + m_B(x)$.
- **Intersección** Sean A y B multiconjuntos, en la intersección de A y B (denotada $A \cap B$), para todo $x \in A \cap B$, $m_{A \cap B}(x) = \min\{m_A(x), m_B(x)\}$, donde *min* es la función que regresa el menor de los dos números.
- **Diferencia** Sean A y B multiconjuntos, en la diferencia de A y B (denotada A/B), para todo $x \in A/B$, $m_{A \cap B}(x) = m_A(x) - m_B(x)$.

Implementación

Utilizaremos la definición de multiconjuntos como conjunto de pares para nuestra implementación, es decir, un multiconjunto será una lista de pares (a,Int).
Lo anterior se refleja en el siguiente tipo de dato:

```
type Multi a = [(a,Int)]
```

No restringiremos el tipo del multiconjunto por lo que utilizaremos una estructura parametrizada, por otro lado, la multiplicidad estará dada únicamente por enteros.

Ejemplo, la función `mElem` que determina si un elemento pertenece a un multiconjunto.

```
--Función mElem
mElem :: (Eq a) => Multi a -> a -> Bool
mElem [] e = False
mElem (x:xs) e = if e == (fst x) then True else
                 mElem xs e

--Ejecución de la función mElem con [(1,2),(2,5),(3,2),(4,2)]
--y 3
*Multiconjuntos> mElem [(1,2),(2,5),(3,2),(4,2)] 3
True
```

Ejercicios

1. **trans**
   Tipo : trans :: trans :: Eq a => [a] -> Multi a
   Especificación : Transforma una lista en un multiconjunto.

   Ejemplos :

```
*Multiconjuntos> trans [1,2,3,4,2,4,1,3,2,2,2]

[(1,2),(2,5),(3,2),(4,2)]
```

2. **mult**
   Tipo : mult :: Eq a => a -> Multi a -> Int
   Especificación : Dado un elemento, regresa la multiplicidad en un multiconjunto.

   Ejemplos :

```
*Multiconjuntos> mult 2 [(1,2),(2,5),(3,2),(4,2)]
5
```

3. **subM**
   Tipo : subM :: Eq a => Multi a -> Multi a -> Bool
   Especificación : Función que diga si un multiconjunto es submulticonjunto de otro.

   Ejemplos :

```
*Multiconjuntos> subM [(2,5),(3,2),(4,2)]
                      [(2,6),(3,8),(4,1)]
False
*Multiconjuntos> subM [(2,5),(4,2)] [(2,6),(3,8),(4,5)]
True
```

4. **igualM**
   Tipo : igualM :: Eq a => Multi a -> Multi a -> Bool
   Especificación : Verifica que dos multiconjuntos sean iguales.
   Para que sean igual deben tener los mismos elementos repetidos el mismo número de veces.

   Ejemplos :

```
*Multiconjuntos> igualM [(3,8),(2,6),(4,5)]
                        [(2,6),(3,8),(4,5)]
True
```

5. **unionM**
   Tipo : unionM :: Eq a => Multi a -> Multi a -> Multi a
   Especificación : Función que realiza la unión entre multiconjuntos.

   Ejemplos :

```
*Multiconjuntos> unionM [(2,1),(4,3),(3,2)] [(3,5),(5,2)]
[(2,1),(4,3),(3,7),(5,2)]
```

6. **interM**
   Tipo : interM :: Eq a => Multi a -> Multi a -> Multi a
   Especificación : Función que realiza la intersección entre multiconjuntos.

   Ejemplos :

```
*Multiconjuntos> interM [(2,1),(4,3),(3,2)] [(3,5),(5,2)]
[(3,2)]
```

7. **difM**

Tipo : difM :: Eq a => Multi a −> Multi a −> Multi a
Especificación : Función que realiza la diferencia entre dos multiconjuntos.

Ejemplos :

```
*Multiconjuntos> difM [(2,3),(4,3),(7,1)]
                      [(3,1),(4,2),(8,1)]
[(2,3),(4,1),(7,1)]
*Multiconjuntos> difM [(2,3),(4,3),(7,1)]
                      [(3,1),(4,5),(8,1)]
[(2,3),(7,1)]
```

8. **agregaM**

Tipo : agregaM :: Eq a => a −> Multi a −> Multi a
Especificación : Agrega un elemento al multiconjunto, si el elemento ya existe, aumenta la multiplicidad, si no, se crea un nuevo par.

Ejemplos :

```
*Multiconjuntos> agregaM 2 [(2,3),(4,3),(7,1)]
[(2,4),(4,3),(7,1)]
*Multiconjuntos> agregaM 8 [(2,3),(4,3),(7,1)]
[(2,3),(4,3),(7,1),(8,1)]
```

9. **agregaML**

Tipo : agregaML :: Eq a => [a] −> Multi a −> Multi a
Especificación : Agrega todos los elementos de la lista al multiconjunto

Ejemplos :

```
*Multiconjuntos> agregaML [3,2,4,52,2,1]
                          [(2,3),(4,3),(7,1)]
[(2,5),(4,4),(7,1),(3,1),(52,1),(1,1)]
*Multiconjuntos> agregaML [4,7,11,2] [(2,3),(4,3),(7,1)]
[(2,4),(4,4),(7,2),(11,1)]
```

10. **eliminaM**

Tipo : eliminaM :: Eq a => a −> Multi a −> Multi a
Especificación : Elimina un elemento en un multiconjunto.
Si m_T (x) > 1 entonces, m_T(x) = m_T(x) - 1.
Si m_T (x) = 1 entonces el elmento se elimina totalmente del multiconjunto pues la multiplicidad cero no está permitida.

Ejemplos :

```
*Multiconjuntos> eliminaM 7 [(2,3),(4,3),(7,1)]
[(2,3),(4,3)]
*Multiconjuntos> eliminaM 4 [(2,3),(4,3),(7,1)]
[(2,3),(4,2),(7,1)]
```

11. **eliminaML**
    Tipo : eliminaML :: Eq a => a -> Multi a -> Multi a
    Especificación : Para todos los elementos de la lista, los elimina del multiconjunto. Si algún elemento no se encuentra en el multiconjunto, simplemente se ignora.

    Ejemplos :

```
*Multiconjuntos> eliminaML [3,2,4,52,2,1]
                           [(2,3),(4,3),(7,1)]
[(2,1),(4,2),(7,1)]
*Multiconjuntos> eliminaML [4,7,11,2] [(2,3),(4,3),(7,1)]
[(2,2),(4,2)]
```

Extras

1. Transforma un multiconjunto en una lista ordinaria.
2. Realiza una función que calcule la diferencia simétrica de dos multiconjuntos.
3. Define una función que realice el producto cartesiano de dos multiconjuntos, el producto cartesiano lo podemos ver como otro multiconjunto, pues los pares pueden ser repetidos.
4. Realiza la instacia de **Show** para los multiconjuntos. Se deberán mostrar los elementos ordenados dentro de la lista.

Cuestionario

1. Investiga otras definiciones de multiconjuntos.
2. ¿Qué pasaría si en la unión utilizaramos el máximo de las multiplicidades?

Práctica 7

Conversiones numéricas

Objetivos

Implementar funciones para convertir números en sus distintas *respresentaciones numéricas* utilizando *listas*, definiciones de tipo `data` y *recursión*.

Preliminares

Existen diferentes sistemas de numeración, cada uno se caracteriza por los símbolos que se utilizan para representar los números, y el valor que tiene cada símbolo. El sistema de númeración que hemos usado siempre es el decimal, este consta de los símbolos $\{0,1,2,3,4,5,6,7,8,9\}$ y el valor de estos, está dado por su posición, las cuales conocemos como unidades, decenas, centenas, etc.
En los sistemas de númeración posicionales[1] utilizaremos un término llamado *base*, la base en un sistema de númeración estará dada por el número de signos que posee dicho sistema, por ejemplo, en el sistema decimal, dado que tenemos diez signos, diremos que está en base 10.
Para hacer explícita la base en la que estamos trabajando usaremos la siguiente notación: $numero_{base}$, por ejemplo, 345_{10}.
Las bases nos sirven para poder otorgarle un valor diferente a cada símbolo dependiendo de su posición, para esto, se multiplicará el i-ésimo digito por la base elevada a la i-ésima potencia ($base^i$) empezando desde 0, de derecha a izquierda, y luego sumaremos el resultado de todas las multiplicaciones, por ejemplo, el número 345_{10} lo obtenemos así:

$$345_{10} = (3 * 10^2) + (4 * 10^1) + (5 * 10^0) \tag{7.1}$$

En computación, nos interesamos por sistemas de numeración que pueda entender una computadora, el sistema por excelencia de la computación es el sistema binario, al igual que el decimal, es un sistema posicional, sólo que este cuenta únicamente con dos símbolos $\{0,1\}$, por lo que al hablar de números en sistema binario, diremos que están en base 2.
Otros sistemas importantes en computación son el sistema octal y el sistema hexadecimal, en el

[1] Los números romanos son un ejemplo de sistemas de numeración no posicional, ya que los símbolos que utiliza siempre tienen el mismo valor.

sistema octal contamos con 8 símbolos {0,1,2,3,4,5,6,7}, mientras que en el sistema hexadecimal contamos con 16 símbolos {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}, en este sistema en particular usamos letras para representar los números 10, 11, 12, 13,14 y 15. Ambos son sistemas posicionales como el decimal y el binario.
Para poder trabajar con números en distintas bases o sistemas de númeración, debemos saber como hacer conversiones entre ellas, partiendo desde el sistema más cómun para nosotros el decimal.

Conversión decimal - base n

Para convertir un número decimal *m* a base *n*, se deben seguir los siguientes pasos:

1. Dividir *m* entre *n*, obteniendo el cociente de la división (*c*).

2. El cociente *c* obtenido en la división anterior se convierte en nuestro decimal *m*.

3. Se repiten el paso uno hasta que el cociente de una división sea cero.

4. Para obtener el número base n, debemos colocar todos los residuos de las divisiones que hicimos, en el orden inverso en el que fueron obtenidos, es decir, el residuo de la última división va a ser el primer dígito del número en base n, de tal forma que el último dígito del número en base n, será el residuo de la primera división.

Ejemplos:

base 2	base 8	base 16
$72_{10} = 1001000_2$	$416_{10} = 640_8$	$1995_{10} = 7CB_{16}$
72 ÷ 2 = 36 Residuo 0	416 ÷ 8 = 52 Residuo 0	1995 ÷ 16 = 124 Residuo 11 = B
36 ÷ 2 = 18 Residuo 0	52 ÷ 8 = 6 Residuo 4	124 ÷ 16 = 7 Residuo 12 = C
18 ÷ 2 = 9 Residuo 0	6 ÷ 8 = 0 Residuo 6	7 ÷ 16 = 0 Residuo 7
9 ÷ 2 = 4 Residuo 1		
4 ÷ 2 = 2 Residuo 0		
2 ÷ 2 = 1 Resicuo 0		
1 ÷ 2 = 0 Residuo 1		

Conversión base n - decimal

Para convertir un número base *n* a decimal se deben seguir los siguientes pasos:

1. Multiplicamos la base *n* elevada a la i-ésima potencia ($base^i$) por el i-ésimo dígito, empezando desde 0, de derecha a izquierda.

2. Sumar el resultado de todas las multiplicaciones.

Ejemplos:

base 2	base 8	base 16
$1001000_2 = 72_{10}$	$640_8 = 416_{10}$	$7CB_{16} = 1995_{10}$
$0 * 2^0 = 0$ $0 * 2^1 = 0$ $0 * 2^2 = 0$ $1 * 2^3 = 8$ $0 * 2^4 = 0$ $0 * 2^5 = 0$ $1 * 2^6 = 64$	$0 * 8^0 = 0$ $4 * 8^1 = 32$ $6 * 8^2 = 384$	$B * 16^0 = 11$ $C * 16^1 = 192$ $7 * 16^2 = 1792$
64 + 8 = 72	32 + 384 = 416	11 + 192 + 1792 = 1995

Conversión binario - octal - binario

Para hacer las conversiones de binario a octal, sin hacer la conversión a decimal como paso intermedio debemos:

1. Agrupar todos los signos del número binario en bloques de tres, empezando de derecha a izquierda, si al tratar de formar el último bloque nos faltan símbolos para que sean tres elementos, podemos agregar ceros que completen el bloque a la izquierda.

2. Una vez formados los bloques convertiremos los tres dígitos correspondientes a decimal, de esta forma, respetando el orden de los bloques obtendremos el número octal correspondiente.

Ejemplo:

Transformar $18_{10} = 10010_2$ a octal. Podemos obtener dos bloques $\underset{\sim}{10}\,\underline{010}$ pero podemos ver que a uno le falta un elemento para ser un bloque de tres elementos, por lo que debemos añadir un cero a la izquierda, por lo que nos quedan los siguientes bloques : $\underset{\sim}{010}\ \underline{010}$. Convertimos cada bloque a su representación decimal, obteniendo: $\underset{\sim}{2}\ \underline{2}$.
Por lo tanto $10010_2 = 22_8 = 18_{10}$

Para convertir un número octal a binario:

1. Cada dígito del número en representación octal lo transformaremos a un número binario de tres elementos, si la representación binaria no tiene tres elementos, agregaremos los ceros que hagan falta a la izquierda hasta que tengamos tres elementos.

2. Respetaremos el orden de los dígitos para obtener el binario.

Ejemplo:

Transformar 56_8 a binario, primero convertimos 6 a binario, lo cual es 110, después convertimos el 5 a binario, 101. Respetando el orden original de los elementos, $56_8 = 101110_2$

Conversión binario - hexadecimal - binario

La conversión de binario a hexadecimal es hacer algo similar que con la transformación a octal, sólo que en lugar de agrupar el número en bloques de tres, lo agruparemos en bloques de cuatro y procederemos de la misma forma. Similarmente, para convertir un número hexadecimal a binario, procederemos de la misma forma que como lo hicimos de octal a binario, sólo que los números binarios que obtengamos tienen que ser de cuatro elementos.

Conversión octal - hexadecimal - octal

Para hacer estas conversiones, será necesario usar los números binarios como intermediario, por lo que para convertir un número octal a hexadecimal, primero transformaremos el número octal a binario y procederemos a convertir el número binario a hexadecimal como vimos anteriormente. Lo mismo ocurrirá para transformar un número hexadecimal a octal, usaremos los números binarios como intermediarios, y procederemos como ya sabemos hacerlo.

Implementación

Los números base 2 y base 8, estarán representados con listas de enteros, teniendo en cuenta que dichos números sólo pueden ser positivos, y en particular, para números en base 2, únicamente serán listas de unos y ceros. Así mismo para números en base 8, sólo serán listas con los enteros del cero al siete.

Para los números base 16, dado que se componen de letras y números y una de las características principales de *Haskell* son las listas homogéneas, no podemos formar los números como en los casos anteriores, por lo que utilizaremos listas de Strings, así para representar al número 1, usaremos al String "1", y así para todos los números, tomando en cuenta que sólo podemos tener números del 1 al 9 y las demás serán las letras correspondientes.
Para que dichas conversiones se realicen con mayor facilidad, estará permitido el uso de las funciones `show` y `read` del *preludio* de *Haskell*.

Los siguientes tipos de datos representan lo que se mencionó anteriormente:

```
Binario = [Int]
Octal = [Int]
Hexa = [String]
```

Ejemplo, la función `binToDec` que convierte un número binario a decimal.

```
-- Función binToDec
binToDec :: [Int] -> Int
binToDec xs = binToDec' xs ((length xs) - 1)
-- Función auxiliar para binToDec
binToDec' :: [Int] -> Int -> Int
binToDec' [] n = 0
binToDec' (x:xs) n = (x*(2^n)) + binToDec' xs (n-1)
```

```
--Ejecución de la función binToDec con [1,1,1]
*Conversiones> binToDec [1,1,1]
7
```

Ejercicios

1. **decToBin**
   Tipo : decToBin :: **Int** −> Binario
   Especificación : Transforma un número a binario. Únicamente transformaremos números positivos, por lo que trabajaremos con el valor absoluto de los números que recibamos.

   Ejemplos :

```
*Conversiones> decToBin 7
[1,1,1]
*Conversiones> decToBin (-15)
[1,1,1,1]
```

2. **decToOct**
   Tipo : decToOct :: **Int** −> Octal
   Especificación : Convierte un número decimal a octal.

   Ejemplos :

```
*Conversiones> decToOct 416
[6,4,0]
*Conversiones> decToOct 456
[7,1,0]
```

3. **octToDec**
   Tipo : octToDec :: Octal −> **Int**
   Especificación : Convierte un número octal a decimal.

   Ejemplos :

```
*Conversiones> octToDec [6,4,0]
416
*Conversiones> octToDec [7,1,0]
456
```

4. **binToOct**
   Tipo : binToOct :: Binario −> Octal
   Especificación : Convierte un número binario a octal, sin pasar por decimales.

   Ejemplos :

```
*Conversiones> binToOct [1,1,1,1,1,1,1,1]
[3,7,7]
*Conversiones> binToOct [1,0,1,0,1,1,0,1]
[2,5,5]
```

5. **octToBin**
   Tipo : octToBin :: Octal -> Binario
   Especificación : Convierte un número octal a binario, sin pasar por decimales.

   Ejemplos :

```
*Conversiones> octToBin [3,6,1]
[0,1,1,1,1,0,0,0,1]
*Conversiones> octToBin [7,4,2]
[1,1,1,1,0,0,0,1,0]
```

6. **decToHexa**
   Tipo : decToHexa :: **Int** -> Hexa
   Especificación : Convierte un número decimal a hexadecimal

   Ejemplos :

```
*Conversiones> decToHexa 1995
["7","C","B"]
*Conversiones> decToHexa 1515
["5","E","B"]
```

7. **hexaToDec**
   Tipo : hexaToDec :: Hexa -> **Int**
   Especificación : Convierte un número de hexadecimal a decimal.

   Ejemplos :

```
*Conversiones> hexaToDec ["7","C","B"]
1995
*Conversiones> hexaToDec ["5","D","A"]
1498
```

8. **binToHexa**
   Tipo : binToHexa :: Binario -> Hexa
   Especificación : Convierte un número binario a hexadecimal, sin pasar por decimales.

   Ejemplos :

```
*Conversiones> binToHexa [1,1,0,1,1,0,0,1,1]
["1","B","3"]
*Conversiones> binToHexa [1,0,0,1,0,0,1,1,0]
["1","2","6"]
```

9. **hexaToBin**
   Tipo : hexaToBin :: [**String**] −> [**Int**]
   Especificación : Convierte un número hexadecimal a binario, sin pasar por decimales.

   Ejemplos :

```
*Conversiones> hexaToBin ["C","3","4"]
[1,1,0,0,0,0,1,1,0,1,0,0]
*Conversiones> hexaToBin ["8","A","F"]
[1,0,0,0,1,0,1,0,1,1,1,1]
```

10. **octToHexa**
    Tipo : octToHexa :: [**Int**] −> [**String**]
    Especificación : Convierte un número octal a hexadecimal, sin pasar por decimales.

    Ejemplos :

```
*Conversiones> octToHexa [6,7,4]
["1","B","C"]
*Conversiones> octToHexa [5,7,3]
["1","7","B"]
```

11. **hexaToOct**
    Tipo : hexaToOct :: [**String**] −> [**Int**]
    Especificación : Convierte un número hexadecimal a octal, sin pasar por decimales.

    Ejemplos :

```
*Conversiones> hexaToOct ["A","4","3"]
[5,1,0,3]
*Conversiones> hexaToOct ["8","C","6","1"]
[1,0,6,1,4,1]
```

Extras

1. Crea una función que verifique según la base que se de, si el número está construido de manera correcta, por ejemplo, un número en base 2, no tiene dentro de su lista números mayores a 1, ni números negativos. La función verificará para cada número si cumple con las condiciones de su base.

2. Realiza una función que transforme cualquier decimal, sea negativo o positivo a binario. Para hacer esto, el primer número de la lista indicará el signo del número, 0 negativo, 1 positivo.

3. Elabora la función que transforme un número binario con distinción de signo (el primer elemento de la lista indica el signo, 0 negativo, 1 positivo), a decimal.

4. Define un tipo de datos que represente a los números hexadecimales.

5. De acuerdo con el tipo de datos que se define en el ejercicio anterior, realiza las conversiones: binario-hexadecimal, hexadecimal-binario, octal-hexadecimal, hexadecimal-octal.

Cuestionario

1. Explica claramente cómo harías una función que transforme un número decimal cualquiera a un número en cualquier base entre 3 y 9. Indica cómo seria la firma de la función y si necesitas de funciones auxiliares.

2. Explica claramente cómo harías una función que hiciera cualquier conversión entre bases de 2 a 9.

3. ¿Es posible hacer cualquier conversión de números entre 2 y 16 con *Haskell*? Explica tu respuesta.

4. Explica por qué en la transformación de octal a binario se necesitan bloques de 3 elementos y en hexadecimal bloques de 4 elementos.

5. Averigua los procedimientos complemento a 1 y complemento a 2.

Práctica 8

Permutaciones de listas

Objetivo

Implementar diferentes funciones para determinar *permutaciones en listas* utilizando *recursión* y *algorítmos de ordenamiento*.

Preliminares

La permutación de una lista será aquella que cuenta con los mismos elementos, pero dichos elementos puden estar en posiciones diferentes dentro de la lista.
Hay cuatro maneras de definir la permutación de una lista:

1. **Eliminación:** Para comprobar que una lista ℓ es permutación de otra lista ℓ_1, bastará eliminar uno a uno los elementos iguales de ambas listas, si al final ambas listas están vacías, entonces podemos decir que ℓ es permutación de ℓ_1.

2. **Número de elementos:** Para comprobar que una lista ℓ es permutación de otra lista ℓ_1, podemos contabilizar el número de repeticiones de cada elemento de las listas, si ambas cuentan con los mismos elementos, y cada uno de ellos aparece el mismo número de veces, entonces podemos decir que ℓ es permutación de ℓ_1.

3. **Ordenamiento:** Para comprobar que una lista ℓ es permutación de otra lista ℓ_1, basta ordenar ambas listas. Si el resultado es el mismo en ambos casos, entonces podemos decir que ℓ es permutación de ℓ_1.

4. **Lista de permutaciones:** Para comprobar que una lista ℓ es permutación de otra lista ℓ_1, podemos obtener el conjunto de todas las posibles permutaciones de ℓ y luego verificar que ℓ_1 se encuentra en ese conjunto, si la encontramos en cuyo caso podemos decir que ℓ es permutación de ℓ_1.

Implementación

Usaremos listas de elementos comparables y ordenables para desarrollar los cuatro casos.

Ejercicios

Deberás implementar la función que decide si una lista es permutación de otra, usando las cuatro definiciones que mencionamos anteriormente.
Para la comprobación por ordenamiento se deberá implementar quickSort.

1. **esPer1**
   Tipo : esPer1 :: (**Ord** a, **Eq** a) =>[a] −> [a] −> **Bool**
   Especificación : Decide si una lista es permutación de otra, utlilizando el método de Eliminación.

   Ejemplos :

```
*Permutaciones> esPer1 [1,3,4,2] [1,2,3,4]
True
*Permutaciones> esPer1 [1,3,4,5] [1,2,3,4]
False
```

2. **esPer2**
   Tipo : esPer2 :: (**Ord** a, **Eq** a) =>[a] −> [a] −> **Bool**
   Especificación : Decide si una lista es permutación de otra, utlilizando el método de Número de elementos.

   Ejemplos :

```
*Permutaciones> esPer2 "hola" "olhas"
False
*Permutaciones> esPer2 "hola" "olha"
True
```

3. **quickSort**
   Tipo : quickSort :: (**Ord** a, **Eq** a) =>[a] −>$ [a]
   Especificación : Ordena una lista usando el algoritmo de quickSort.

   Ejemplos :

```
*Permutaciones> quickSort [9,3,2,4,1]
[1,2,3,4,9]
*Permutaciones> quickSort "hola"
"ahlo
```

4. **esPer3**
   Tipo : esPer3 :: (**Ord** a, **Eq** a) =>[a] −> [a] −> **Bool**
   Especificación : Decide si una lista es permutación de otra con el método de ordenamiento QuickSort.

   Ejemplos

```
*Permutaciones> esPer3 "hola" "olhas"
False
*Permutaciones> esPer3 "hola" "olha"
True
```

5. **esPer4**
   Tipo : esPer4 :: (**Ord** a, **Eq** a) =>[a] −> [a] −> **Bool**
   Especificación : Decide si una lista es permutación de otra, utilizando el método de lista de permutaciones.

   Ejemplos :

```
*Permutaciones> esPer4 "batman" "manbat"
True
*Permutaciones> esPer4 [1,4,2,3] [4,2,2,5]
False
```

Extras

1. Realiza la comprobación por ordenamiento usando mergeSort.
2. Elabora la instancia de Eq, con cualquier función de los ejercicios anteriores.
3. Define la función que determine si un elemento $a \in A$ pertenece a una lista ℓ_A.
4. Usando la función anterior, verifica que una lista ℓ es permutación de otra lista ℓ_1 si todos los elementos de ℓ también pertenecen a ℓ_1. Debes tener cuidado con los elementos repetidos.

Cuestionario

1. Explica a grandes rasgos la complejidad de cada manera de comprobación.
2. Escribe otra forma de comprobar si una lista es permutación de otra.
3. Demuestra que la relación *permutación* es una relación de equivalencia.
4. Sean las listas ℓ y ℓ_1, demuestra que si ℓ es permutación de ℓ_1 entonces para toda y, (y:ℓ) es permutación de (y:ℓ_1).
5. Sean las listas ℓ y ℓ_1, demuestra que si ℓ es permutación de [] entonces ℓ = [].

Práctica 9

Notación prefija, infija y sufija

Objetivo

Implementar la traducción de *gramática de expresiones aritméticas* a sus diferentes *notaciones (prefija, infija y sufija)* utilizando definiciones de tipo `data` y *recursión*.

Notaciones

Notación infija

La notación infija es la que usamos normalmente, tanto en álgebra como en lógica y computación. Se caracteriza por poner los operadores entre los operandos, por ejemplo para sumar 2 mas 2, lo más cómun es escribir $2+2$, con el operador $+$ entre los operandos 2 y 2.
Para la evaluación de expresiones en notación infija, hay un pequeño detalle, puede que queramos que una operación se realice antes que otra, para esto debemos tener una precedencia de operadores y en caso de tener operadores con la misma precedencia, debemos desambiguar la expresión con el uso de paréntesis.

La precedencia usual de operadores aritméticos es la siguiente:

Prioridad	**Operación**
1	Paréntesis
3	Multiplicaciones y divisiones
4	Sumas y restas

Un ejemplo de expresión aritmética bien formada y sin ambigüedad de precedencia es la siguiente: $e = (2*3+8)/2$. Por el contrario la expresión : $f = 2*3/2$ causa problemas pues la multiplicación y la división tienen la misma precedencia, entonces, necesitamos usar paréntesis para arreglar dicha ambiguedad.

Notación prefija

La notación prefija, también conocida como notación polaca, se caracteriza por poner sus operadores a la izquierda de sus operandos. Por ejemplo para la expresión infija 2 + 2, su correspondiente expresión en notación prefija es + 2 2.

La característica de esta notación es que los paréntesis no son necesarios y la precedencia de los operadores está dada por el orden en el que aparecen en la expresión, esto es posible dado que sabemos el número de argumentos de cada operador. Por ejemplo, la versión prefija de $e = (2*3+8)/2$ es: $exp = /+*2\ 3\ 8\ 2$
En *Haskell* este tipo de notación es la más utilizada, tanto para definir tipos de datos, como para funciones y operadores, aunque podemos definir también operaciones infijas.

Notación posfija

La notación postfija, tambien conocida como polaca inversa, se caracteriza por poner sus operadores a la derecha de sus operandos. Por ejemplo la versión posfija de la expresión infija 2 + 2, es 2 2 +.
En esta notación tampoco necesitamos paréntesis para desambiguar la expresión. La expresión posfija de $e = (2*3+8)/2$ es : $epr = 2\ 3*8+2\ /$

Implementación

Vamos a implementar las distintas notaciones, así como su proceso de evaluación, Las expresiones las recibiremos como *String*, pero haremos una tranformación a los siguientes tipos de datos:

```
data Op = Suma | Resta | Multi | Div deriving (Show,Eq)
data Par = L | R deriving (Show,Eq)
data Token = TO Op | TP Par | NumE Int deriving (Show,Eq)
```

Los tipos anteriores representan a los operadores aritméticos, los paréntesis izquierdo y derecho y a un token que puede ser un operador o un paréntesis o un números. A partir de ellos, las diferentes notaciones estarán dadas por los siguientes tipos :

```
type Prefijo = [Token]
type Infijo = [Token]
type Posfijo = [Token]
```

La evaluación de expresiones dependiendo de la notación en la que se encuentren son las siguientes :

- **Expresiones infijas**

1. Leemos la expresión de izquierda a derecha.
2. Si leemos un número lo guardamos al principio de la lista A.
3. Si leemos un paréntesis izquierdo, buscamos el paréntesis derecho que lo cierre y evaluamos todo lo que está entre ellos poniendo el resultado al principio de la lista A.
4. Si leemos un operador, inicialmente lo guardamos en la lista B, pero si el primer elemento de la lista es un operador de mayor prioridad, eliminamos ese operador de la lista, hacemos la evaluación corresponde y procedemos a guardar el nuevo operador al principio de la lista.

Como podemos ver, para la evaluación infija son necesarias dos listas.

- **Expresiones prefijas**

1. Leemos la expresión de derecha a izquierda.
2. Si leemos un número lo guardamos al principio de una lista.
3. Cuando leamos un operador, sacamos los dos primeros elementos de la lista, realizamos la operación y el resultado lo guardamos al principio de la lista.

- **Expresiones posfijas**

1. Leemos la expresión de izquierda a derecha.
2. Si leemos un número lo guardamos al principio de una lista.
3. Cuando leamos un operador, sacamos los dos primeros elementos de la lista, realizamos la operación y el resultado lo guardamos al principio de la lista.

Ejercicios

1. **transformaPr**

   Tipo : transformaPr :: **String** −> Prefijo
   Especificación : Transforma la cadena de entrada a una expresión prefija representada con tokens.
   Nota: Supondremos que la cadena de entrada corresponde a una expresión bien construida en notación prefija.

   Ejemplos :

```
*Notaciones> transformaPr "+ 1 + 3 4"
[TO Suma,NumE 1,TO Suma,NumE 3,NumE 4]
*Notaciones> transformaPr "* 4 / 3 + 2 3"
[TO Multi,NumE 4,TO Div,NumE 3,TO Suma,NumE 2,NumE 3]
```

2. **transformaIn**

Tipo : transformaIn :: **String** −> Infijo

Especificación : Transforma la cadena de entrada a una expresión infija representada con tokens.
Nota: Supondremos que la cadena de entrada ya cuenta con una expresión en notación infija.

Ejemplos :

```
*Notaciones> transformaIn "3 + 8 * ( 4 + 5 )"
[NumE 3,TO Suma,NumE 8,TO Multi,TP L,
 NumE 4,TO Suma,NumE 5,TP R]

*Notaciones> transformaIn "3 * 4 + 8 / 2"
[NumE 3,TO Multi,NumE 4,TO Suma,NumE 8,TO Div,NumE 2]
```

3. **transformaPs**

Tipo : transformaPs :: **String** −> Posfijo

Especificación : Transforma la cadena de entrada a una expresión posfija representada con tokens.
Nota: Supondremos que la cadena de entrada ya cuenta con una expresión en notación posfija.

Ejemplos :

```
*Notaciones> transformaPs " 5 1 2 + 4 * + 3 -"
[NumE 5,NumE 1,NumE 2,TO Suma,NumE 4,TO Multi,
                    TO Suma,NumE 3,TO Resta]
*Notaciones> transformaPs "3 8 4 5 + * +"
[NumE 3,NumE 8,NumE 4,NumE 5,TO Suma,TO Multi,TO Suma]
```

4. **evalPr**

Tipo : evalPr :: Prefijo −> **Int**

Especificación : Evalua una expresión en notación Prefija.

Ejemplos :

```
*Notaciones> evalPr [TO Suma,NumE 1,TO Suma,NumE 3,NumE 4]
8
*Notaciones> evalPr [TO Multi,NumE 4,TO Div,NumE 3,TO Suma,
                                        NumE 2,NumE 3]
4
```

5. **evalIn**

Tipo : evalIn :: Prefijo −> **Int**

Especificación : Evalua una expresión en notación Infija.

Ejemplos :

```
*Notaciones> evalIn [NumE 3,TO Multi,NumE 4,TO Suma,
                                NumE 8,TO Div,NumE 2]
16
*Notaciones> evalIn [NumE 3,TO Suma,NumE 8,TO Multi,TP L,
                              NumE 4,TO Suma,NumE 5,TP R]
75
```

6. **evalPs**
   Tipo : evalPs :: Prefijo −> **Int**
   Especificación : Evalua una expresión en notación Posfijo.

   Ejemplos :

```
*Notaciones> evalPs [NumE 5,NumE 1,NumE 2,TO Suma,NumE 4,
                     TO Multi,TO Suma,NumE 3,TO Resta]
14
*Notaciones> evalPs [NumE 3,NumE 8,NumE 4,NumE 5,TO Suma,
                        TO Multi,TO Suma]
75
```

Ejercicios extra

1. Define una gramática para expresiones prefijas con un tipo de datos de *Haskell*.
2. Define una gramática para expresiones infijas con un tipo de datos de *Haskell*.
3. Realiza una función que transforme de prefijo a infijo con los datas que definiste anteriormente.
4. Realiza una función que transforme de infijo a prefijo con los datas que definiste anteriormente.
5. Amplia la definición para que acepte exponentes y raíces. Cambiando el tipo del número a Float.

Cuestionario

1. ¿Es posible definir una gramática sufija usando los data de *Haskell*? ¿Por qué?
2. ¿Qué pasa si agrego números con signo en las expresiones?, ¿cómo se complican las implementaciones?

Práctica 10

Enteros y racionales como pares

Objetivo

Implementar una representación de los número *enteros* y *racionales* vistos como pares, utilizando *recursión* y definiciones de tipo `type`.

Preliminares

En ciertas ocasiones nos interesa poder agrupar en una misma clase a todos los objetos que cumplan una propiedad y tratar como iguales a cualesquiera individuos de una clase particular, para esto necesitamos una clase especial de relación que respete las propiedades escenciales de la igualdad. A estas relaciones les llamamos relaciones de equivalencia.[1]

Definición

Una relación R en un conjunto A es de equivalencia si cumple con tres propiedades: [3]

- Es reflexiva.
- Es simétrica, es decir $(a,b) \in R \Rightarrow (b,a) \in R$.
- Es transitiva, es decir $(a,b) \in R \wedge (b,c) \in R \Rightarrow (a,c) \in R$.

Clase de equivalencia

Una vez que definimos relaciones de equivalencia, podemos usarlas para identificar elementos que sean equivalentes como uno mismo, para esto definimos clases de equivalencia de la siguiente manera:

Si R es una relación de equivalencia en el conjunto A, $x \in A$, definimos a la clase de equivalencia de x, $[x]_R$ por: [3]

$$[x]_R = \{y \in A \mid xRy\}$$

Enteros

En el conjunto de números naturales $\mathbb{N}$, podemos efectuar las operaciones de suma y multiplicación, pero cuando queremos resolver la ecuación $a = b + x$ no siempre podemos encontrar una solución en $\mathbb{N}$, por lo cual debemos extender nuestro conjunto a otro donde dicha ecuación pueda resolverse.
Para toda ecuación de la forma arriba mencionada se obtiene $\mathbb{Z}$, el conjunto de los números Enteros.
Si queremos obtener el valor de x en la ecuación que teníamos en un inicio $a = b + x$, tendríamos que $x = a - b$, notemos que ambas ecuaciones están determinadas por la pareja (a,b) y dicha pareja puede representar a cualquier entero x, mediante $x = a - b$.

Racionales

La motivación para los números racionales, es básicamente lo mismo que con los números enteros, tenemos la ecuación $a = b * x$, dicha ecuación no siempre tiene solución en los números enteros, por lo que se tuvo que crear el conjunto $\mathbb{Q}$ de los números racionales para poder tener una solución, de esta forma podríamos obtener el valor de x de la forma $x = a/b$.
Nuevamente ambas ecuaciones están dadas por la pareja (a,b), entonces, en los números racionales, la pareja (a,b) representará al racional x, mediante la ecuación $x = a/b$.

Implementación

La implementación de ambas representaciones tanto de enteros como racionales, estará dada por pares de tipo Int, `(Int,Int)`, para evitar confunciones en la práctica renombraremos estos pares con las etiquetas Entero y Racional.

```
-- Tipo para los enteros
type Entero = (Int,Int)
--Tipo para los racionales
type Racional = (Int,Int)
```

Ejemplo, la función `toInt` que transforma un Entero a un Int de *Haskell*.

```
-- Función toInt
toInt :: Entero -> Int
toInt (a,b) = a-b

--Ejecución de toInt con (-4,10)
*EntRel> toInt (-4,10)
-14
```

Ejercicios

Enteros

1. **toEntero**
   Tipo : **toInt** :: **Int** −> Entero
   Especificación : Convierte un Int a un Entero

   Ejemplos :

```
*EntRel> toEntero 19
(19,0)
*EntRel> -9
(0,9)
```

2. **sumaE**
   Tipo : sumaE :: Entero −> Entero −> Entero
   Especificación : Función que realiza la suma de dos Enteros.

   Ejemplos :

```
*EntRel> sumaE (8,-4) (9,3)
(17,-1)
*EntRel> sumaE (4,2) (9,11)
(13,13)
```

3. **restaE**
   Tipo : restaE :: Entero −> Entero −> Entero
   Especificación : Función que realiza la resta de dos Enteros.

   Ejemplos :

```
*EntRel> restaE (-18,3) (6,1)
(-24,2)
*EntRel> restaE (9,11) (3,4)
(6,7)
```

4. **multiE**
   Tipo : multiE :: Entero −> Entero −> Entero
   Especificación : Función que realiza la multiplicacion de dos Enteros

   Ejemplos :

```
*EntRel> multiE (8,3) (9,4)
(84,59)
*EntRel> multiE (2,-5) (-9,5)
(-43,55)
```

5. **divE**
   Tipo : divE :: Entero −> Entero −> Entero
   Especificación : Función que realiza la división entera de dos Enteros

   Ejemplos :

```
*EntRel> divE (18,3) (8,5)
(5,0)
*EntRel> divE (-14,4) (6,-3)
(-2,0)
```

6. **htE**
   Tipo : htE :: Entero −> Entero −> **Bool**
   Especificación : Función que dados dos enteros, nos dice si el primero es mayor que el segundo

   Ejemplos :

```
*EntRel> htE (23,19) (7,3)
False
*EntRel> htE (7,1) (15,14)
True
```

7. **ltE**
   Tipo : ltE :: Entero −> Entero −> **Bool**
   Especificación : Función que dados dos enteros, nos dice si el primero es menor que el segundo

   Ejemplos :

```
*EntRel> ltE (9,2) (13,10)
False
*EntRel> ltE (28,29) (4,2)
True
```

8. **eqE**
   Tipo : eqE :: Entero −> Entero −> **Bool**
   Especificación : Función que dados dos enteros, nos dice si son iguales

   Ejemplos :

```
*EntRel> eqE (9,4) (11,6)
True
*EntRel> eqE (9,4) (12,4)
False
```

Racionales

1. **toFloat**
   Tipo : toFloat :: Racional -> **Float**
   Especificación : Convierte un Racional a un Float

   Ejemplos :

```
*EntRel> toFloat (-8,3)
-2.6666667
*EntRel> toFloat (2,3)
0.6666667
```

2. **inverso**
   Tipo : htE :: Racional -> Racional -> **Bool**
   Especificación : Función que obtiene el inverso de un número racional.

   Ejemplos :

```
*EntRel> inverso (-4,-4)
(4,-4)
*EntRel> inverso (4,6)
(-4,6)
```

3. **sumaR**
   Tipo : sumaR :: Racional -> Racional -> Racional
   Especificación : Función que realiza la suma de dos Racionales.

   Ejemplos :

```
*EntRel> sumaR (1,2) (1,4)
(6,8)
*EntRel> sumaR (6,2) (4,2)
(20,4)
```

4. **restaR**
   Tipo : restaR :: Racional -> Racional -> Racional
   Especificación : Función que realiza la resta de dos Racionales.

   Ejemplos :

```
*EntRel> restaR (1,2) (1,4)
(2,8)
*EntRel> restaR (6,2) (4,2)
(4,4)
```

5. **multiR**
   Tipo : multiR :: Racional -> Racional -> Racional
   Especificación : Función que realiza la multiplicacion de dos Racionlaes

Ejemplos :

```
*EntRel> multiR (6,2) (4,2)
(24,4)
*EntRel> multiR (1,2) (1,4)
(1,8)
```

6. **divR**
   Tipo : divE :: Racional −> Racional −> Racional
   Especificación : Función que realiza la división entera de dos Racionales

Ejemplos :

```
*EntRel> divR (6,3) (4,2)
(12,12)
*EntRel> divR (1,2) (1,4)
(4,2)
```

7. **htR**
   Tipo : htR :: Racional −> Racional −> **Bool**
   Especificación : Función que dados dos racionales, nos dice si el primero es mayor que el segundo

Ejemplos :

```
*EntRel> htR (1,2) (1,4)
True
*EntRel> htR (4,2) (6,2)
False
```

8. **ltR**
   Tipo : ltR :: Racional −> Racional −> **Bool**
   Especificación : Función que dados dos racionales, nos dice si el primero es menor que el segundo

Ejemplos :

```
*EntRel> ltR (1,2) (1,4)
False
*EntRel> ltR (4,2) (6,2)
True
```

9. **eqR**
   Tipo : eqR :: Racional −> Racional −> **Bool**
   Especificación : Función que dados dos racionales, nos dice si son iguales

Ejemplos :

```
*EntRel> eqR (8,4) (12,3)
True
*EntRel> eqR (8,4) (12,4)
False
```

Para las funciones de multiplicación y división de enteros y racionales, no podrán usar las funciones ya definidas en *Haskell*, en ambos casos deberán implementar de manera recursiva sus funciones.

Extras

1. Realiza una función que decida si dos Enteros pertenecen a la misma clase de equivalencia. Estarán relacionados si ambos Enteros representan al mismo número.
2. Realiza una función que decida si dos Racionales pertenecen a la misma clase de equivalencia.
3. Realiza una función que calcule el logaritmo base 2 de un entero.
4. Realiza una función que calcule el máximo común divisor de dos Enteros
5. Realiza una función que reduzca un número racional a su mínima forma.
6. Realiza la instancia de show para los tipos Entero y Racional.
7. Realiza una implementación de números complejos como pares.

Cuestionario

1. Nombra 3 propiedades que cumplen las operaciones en los enteros y los racionales.
2. Nombra 3 propiedades que cumplen las operaciones en los racionales.
3. Demuestra una de estas propiedades en los enteros.
4. Demuestra una de estas propiedades en los racionales.

Práctica 11

Conjuntos como funciones

Objetivo

Implementar *conjuntos* utilizando *funciones de orden superior*.

Preliminares

Podemos ver a los conjuntos como una función booleana, la cual a partir de un elemento regresa True si está en el conjunto, False en otro caso.
Por ejemplo, el conjunto de los números pares puede verse como la función:

$$\text{f(x)} \begin{cases} True, \ Si \ x \ es \ par. \\ False, \ En \ otro \ caso. \end{cases}$$

Otro ejemplo sería el conjunto de los números múltiplos de cinco.

$$\text{g(x)} \begin{cases} True, \ Si \ (x \ mod \ 5) = \ 0. \\ False, \ En \ otro \ caso. \end{cases}$$

Si quisieramos unir los conjuntos anteriores, tendriamos el siguiente conjunto:

$$\text{h(x)} \begin{cases} True, \ Si \ x \ es \ par. \\ True, \ Si \ (x \ mod \ 5) = \ 0. \\ False, \ En \ otro \ caso. \end{cases}$$

Las demás operaciones de conjuntos se calculan de manera similar para esta estructura.

Implementación

Utilizaremos el siguiente tipo para representar a los conjuntos como funciones como las que explicamos anteriormente.

```
type Set = Int -> Bool
```

Esta definición es posible gracias a que un tipo básico de *Haskell* es el tipo función, por lo que no hay ningún problema al usar una función con definición de tipos **type**.

Por simplicidad vamos a utilizar solamente conjuntos de enteros. La mayoría de las funciones que se harán en esta práctica serán funciones de orden superior, ya que se reciben y se regresan conjuntos y éstos a su vez son funciones.
El truco para la implementación de los conjuntos es representarlos mediante su función característica.
Por ejemplo para agregar un nuevo elemento, debemos verificar si ese elemento ya se encontraba en el conjunto, de ser así no hacemos nada, en caso contrario, debemos regresar una función que regrese True con ese nuevo elemento y con todos los demás siga preguntando al conjunto original. Esta función quedaría definida de la siguiente manera:

```
--Función agregar
agregar :: Int -> Set -> Set
agregar a f = if f a then f else g
         where g x  = if x == a then True else f x

{-- Agregar el número 7 al conjunto de los números
    divisibles entre 5.
--}
agregar 7 (\x -> ((mod x 5) == 0))
```

Haskell no sabrá como mostrar el ejemplo anterior ya que las funciones no pertenecen a la clase **Show**, por lo que usaremos la función `elemS` [1] para visualizar el resultado de nuestras funciones de la siguiente manera:

```
--Ejemplo sin agregar 7 al conjunto.
*ConjuntosF> elemS 7 (\x -> ((mod x 5) == 0))
False
--Ejemplo agregando 7 al conjunto.
*ConjuntosF> elemS 7 $ agregar 7 (\x -> ((mod x 5) == 0))
True
```

Ejercicios

1. **pares**
   Tipo : pares :: Set
   Especificación : Define el conjunto de los enteros pares

   Ejemplos :

```
*ConjuntosF> pares 2
True
*ConjuntosF> pares 3
```

[1] Suponemos que la función está definida para usarla en el ejemplo, pero el alumno deberá implementarla como ejercicio.

```
False
```

2. **mCinco**
   Tipo : mCinco :: Set
   Especificación : Define el conjunto de los enteros divisibles entre 5

   Ejemplos :

```
*ConjuntosF> mCinco 15
True
*ConjuntosF> mCinco 8
False
```

3. **elemS**
   Tipo : elemS :: **Int** −> Set −> **Bool**
   Especificación : Decide si un elemento está en el conjunto dado.

   Ejemplos :

```
*ConjuntosF> elemS 12 mCinco
False
*ConjuntosF> elemS 25 mCinco
True
```

4. **eliminar**
   Tipo : eliminar :: **Int** −> Set −> Set
   Especificación : Elimina un elemento del conjunto dado.
   Para comprobar que un elemento se ha eliminado del conjunto, utilizaremos la función elemS.

   Ejemplos :

```
-- Ejemplo sin eliminar 12
*ConjuntosF> elemS 12 pares
True
--Ejemplo eliminando 12
*ConjuntosF> elemS 12 $ eliminar 12 pares
False
```

5. **complemento**
   Tipo : complemento :: Set −> Set
   Especificación : Obtiene el complemento del conjunto dado.
   Para comprobar la operación complemento utilizaremos la función elemS.

   Ejemplos :

```
--Ejemplo sin complemento
*ConjuntosF> elemS 30 pares
True
--Ejemplo con complemento
*ConjuntosF> elemS 30 $ complemento pares
False
```

6. **unionC**

   Tipo : unionC :: Set −> Set −> Set
   Especificación : Realiza la unión de dos conjuntos.
   Para comprobar la operación unión utilizaremos la función elemS.

   Ejemplos :

```
*ConjuntosF> elemS 12 pares
True
*ConjuntosF> elemS 12 mCinco
False
*ConjuntosF> elemS 12 $ unionC pares mCinco
True
```

7. **inter**

   Tipo : inter :: Set −> Set −> Set
   Especificación : Realiza la intersección de dos conjuntos.
   Para comprobar la operación intersección utilizaremos la función elemS.

   Ejemplos :

```
*ConjuntosF> elemS 10 $ inter pares mCinco
True
*ConjuntosF> elemS 12 $ inter pares mCinco
False
*ConjuntosF> elemS 15 $ inter pares mCinco
False
```

8. **difer**

   Tipo : difer :: Set −> Set −> Set
   Especificación : Realiza la diferencia de dos conjuntos.
   Para comprobar la operación diferencia utilizaremos la función elemS.

   Ejemplos :

```
*ConjuntosF> elemS 10 $ difer pares mCinco
False
*ConjuntosF> elemS 12 $ difer pares mCinco
True
```

```
*ConjuntosF> elemS 15 $ difer pares mCinco
False
```

9. **diferSim**
   Tipo : diferSim :: Set -> Set -> Set
   Especificación : Realiza la diferencia simétrica de dos conjuntos.
   Para comprobar la operación diferencia simétrica utilizaremos la función elemS.

   Ejemplos :

```
*ConjuntosF> elemS 15 $ diferSim pares mCinco
True
*ConjuntosF> elemS 10 $ diferSim pares mCinco
False
*ConjuntosF> elemS 12 $ diferSim pares mCinco
True
```

Extras

1. Define las funciones usando lambdas de *Haskell*. Si usaste lambdas originalmente, ahora define tus funciones usando *where*.

2. Realiza el producto cartesiano con este tipo de conjuntos.

Cuestionario

1. ¿Es posible realizar los ejercicios usando `Let`? Explica tu respuesta.

2. Investiga 5 funciones clásicas de orden superior. Explica qué es lo que hacen.

3. Investiga si se puede definir alguna otra estructura como una función.

PARTE III

RESPUESTAS

Respuesta práctica: Gramáticas

```
{-
Módulo      :  Gramáticas
Descripción :  Módulo correspondiente a la práctica de
               gramáticas.
Copyright   :  (c) <Daniela Calderón Pérez>
-}

module Gramáticas where

--Tipos para las gramáticas aritméticas.
data ExpA  = V Vars | Cons Int | S Signo ExpA | OA Oper ExpA ExpA |
             Par ExpA | OperL ExpL ExpL deriving (Show,Eq)

data ExpL = VL Vars | T | F | ParL ExpL | Not ExpL |
            OL OperL ExpL ExpL deriving (Show,Eq)

data Vars = X Int deriving (Show,Eq)

data Signo = Pos | Neg deriving (Show,Eq)

data Oper = Suma | Resta | Mul | Div deriving (Show,Eq)

data OperL = And | Or deriving (Show,Eq)


--Tipos para la gramática de paréntesis balanceados.

data Par = Vacio | LP R Par deriving (Show,Eq)
data R = RP | LP2 R R deriving (Show,Eq)

--Ejemplos de cadenas.

--()()()
```

```
ej1 = LP RP (LP RP (LP RP Vacio))
--(()())()
ej2 = LP (LP2 RP (LP2 RP RP)) (LP RP Vacio)
--()(())()
ej3 = LP RP (LP (LP2 RP RP) (LP RP Vacio))

--Función que cuenta el número de paréntesis en una cadena.
contador :: Par -> Int
contador Vacio = 0
contador (LP r p) = 1 + (cuentaR r) + (contador p)

--Auxiliar de contador.
cuentaR :: R -> Int
cuentaR RP = 1
cuentaR (LP2 r1 r2) = 1+ (cuentaR r1) + (cuentaR r2)

-Función que convierte la gramática Par a String.
showPar :: Par -> String
showPar Vacio = ""
showPar (LP r p) = "(" ++ (showR r) ++ (showPar p)

--Auxiliar de showPar.

showR :: R -> String
showR RP = ")"
showR (LP2 r1 r2) = "(" ++ (showR r1) ++ (showR r2)

--Instancias de la clase show para cadenas.

instance Show (Par) where show p = showPar p

instance Show (R) where show r = showR r
```

Respuesta práctica: Tablas de verdad

```
{-
Módulo      :  Tverdad
Descripción :  Módulo correspondiente a la práctica de
               tablas de verdad.
Copyright   :  (c) <Daniela Calderón Pérez>
-}

module Tverdad where

--Tipos para el módulo.
data Prop = T | F | V NVar | And Prop Prop | Or Prop Prop
          | Not Prop | Imp Prop Prop
          | Eq Prop Prop deriving (Eq,Show)

data NVar = P Int deriving (Eq,Show)

type Renglon = ([Bool],Bool)

type Tabla = [Renglon]

--Función que cuenta las variables de una fórmula.
vars :: Prop -> Int
vars T = 0
vars F = 0
vars (V n) = 1
vars (And p q) = (vars p) + (vars q)
vars (Or p q)  =  (vars p) + (vars q)
vars (Not p) = vars p
vars (Imp p q) = (vars p) + (vars q)
vars (Eq p q) = (vars p) + (vars q)
```

```
--Función que evalúa una fórmula en un estado.
eval :: [Bool] -> Prop -> Bool
eval l T = True
eval l F = False
eval l (V (P n)) = find l n
eval l (And p q) = (eval l p) && (eval l q)
eval l (Or p q) =  (eval l p) || (eval l q)
eval l (Not p) = not (eval l p)
eval l (Imp p q) = implica (eval l p) (eval l q)
eval l (Eq p q) = (eval l p) == (eval l q)


--Función que construye el primer tipo de tabla.
cons1 :: Prop -> Tabla
cons1 p = let v = vars p in
          [((conv x v),(eval (conv x v) p)) | x <- [0..((2^v)-1)]]

--Función que construye el segundo tipo de tabla.
cons2 :: Int -> (Int->Bool) -> Tabla
cons2 n f = if n < 2 then error "numero no valido"
            else [((conv x n), (f x))| x <- [0..((2^n)-1)]]
```

Respuesta práctica: Tableaux

```
{-
Módulo      :  Tableaux
Descripción :  Módulo correspondiente a la práctica de
                        Tableaux.
Copyright   :  (c) <Daniela Calderón Pérez>
-}

module Tableaux where

--Tipos para el modulo
type Var = String
data LProp = F | T | VarP Var | Conj LProp LProp |
             Disy LProp LProp | Imp LProp LProp |
             Equiv LProp LProp | Neg LProp deriving (Show,Eq)

data Tableaux = Hoja [LProp] | Alpha [LProp] Tableaux |
                Beta [LProp] Tableaux Tableaux deriving (Show,Eq)

-- Función que determina si todas las fórmulas de una lista
-- son literales.
literales :: [LProp] -> Bool
literales [] = True
literales (x:xs) = if literal x then literales xs
                   else False

--Auxiliar para literales.
literal :: LProp -> Bool
literal F = True
literal T = True
literal (VarP v) = True
literal (Neg (Neg v)) = False
literal (Neg v) = literal v
literal _ = False
```

```
--Función que determina si una fórmula puede expandirse
  con reglas alpha.
alpha :: LProp -> Bool
alpha (Conj p q) = True
alpha (Neg (Disy p q)) = True
alpha (Neg (Imp p q)) = True
alpha (Equiv p q) = True
alpha _ = False

--Función que determina si una fórmula puede expandirse
-- con reglas beta.
beta :: LProp -> Bool
beta (Disy p q) = True
beta (Neg (Conj p q)) = True
beta (Imp p q) = True
beta (Neg (Equiv p q)) = True
beta _ = False

--Función que hace la expansion alpha.
expAlpha :: [LProp] -> LProp -> [LProp]
expAlpha l r@(Conj p q) = [p,q] ++ (elimina r l)
expAlpha l r@(Neg (Disy p q)) = (Neg p):(Neg q):(elimina r l)
expAlpha l r@(Neg (Imp p q)) = p:(Neg q):(elimina r l)
expAlpha l r@(Equiv p q) = (Imp p q):(Imp q p):(elimina r l)

--Función que hace la expansion sigma.
expSigma :: [LProp] -> LProp -> [LProp]
expSigma l r = case r of
                      (Neg (Neg v)) -> v:nl
                      (Neg T) -> F:nl
                      (Neg F) -> T:nl
               where nl = elimina r l

--Función que construye un Tableau
consTableaux :: LProp -> Tableaux
consTableaux p = arT [p]

--Auxiliar de consTableaux
arT :: [LProp] -> Tableaux
arT p = if literales p then Hoja p
        else let q = nextF p in
          if alpha q then Alpha p (arT (expAlpha p q))
          else if beta q then
          Beta p (arT (fst(expBeta p q))) (arT (snd(expBeta p q)))
               else arT (expSigma p q)
```

```
--Función que determina si una fórmula es tautologia.
tautologia :: LProp -> Bool
tautologia p = cierra (consTableaux (Neg p))
```

Respuesta práctica: Mapas de Karnaugh

```
{- |
Módulo      :  Mapas
Descripción :  Módulo correspondiente a la práctica de
               mapas de Karnaugh.
Copyright   :  (c) <Daniela Calderón Pérez>
-}

module Mapas where

--Tipos para los mapas de Karnaugh
data Var = X | Y deriving (Show,Eq)

data Lit = V Var | Neg Var deriving (Show,Eq)

type KMap = [Bool]

-- Mapa vacío
mk = [False,False,False,False]

-- Lista con todos los posibles bloques que puede tener el mapa

bloq = [[0,2],[0,1],[1,3],[2,3],[0],[1],[2],[3]]

a = Sum (Prod (L (Neg X)) (L (V Y))) (Prod (L (V X)) (L (V Y)))

b = Sum (Prod (L (Neg X)) (L (V Y))) (Prod (L (V X)) (L (Neg Y)))

c = Sum (Prod (L (V X)) (L (Neg Y)))
        (Sum (Prod (L (Neg X)) (L (V Y)))
             (Prod (L (Neg X)) (L (Neg Y))))

--Función que construye el mapa.
consMap :: FBool -> KMap
```

```
consMap f = asigMin2 f mk

--Auxiliar de consMap.
asigMin2 :: FBool -> KMap -> KMap
asigMin2 p@(Prod a b) l = inserta (mint p) l

asigMin2 (Sum a b) l = if (mint a) < 4
                       then asigMin2 b (inserta (mint a) l)
                       else asigMin2 a (inserta (mint b) l)

--Auxiliar de consMap.
inserta :: Int -> KMap -> KMap
inserta 0 (x:xs) = True:xs
inserta n (x:xs) = x:(inserta (n-1) xs)

--Función que nos dice el número de mintérmino que le corresponde
--a cada fórmula.
mint :: FBool -> Int
mint (Prod (L (V X)) (L (V Y))) = 0
mint (Prod (L (V X)) (L (Neg Y))) = 1
mint (Prod (L (Neg X)) (L (V Y))) = 2
mint (Prod (L (Neg X)) (L (Neg Y))) = 3
mint _ = 5

--Función que minimiza bloques.
fMin :: FBool -> FBool
fMin f = let b = bloques (consMap f) in
         if b == [] then f
         else minAux b

--Auxiliar de la función fMin.
minAux :: [[Int]] -> FBool
minAux [] = error "no definido"
minAux (x:[]) = minM x
minAux (x:xs) = Sum (minM x) (minAux xs)

--Auxiliar de la función fMin.
minM :: [Int] -> FBool
minM [0,2] = (L (V Y))
minM [1,3] = (L (Neg Y))
minM [2,3] = (L (Neg X))
minM [0,1] = (L (V X))
minM [0] = (Prod (L (V X)) (L (V Y)))
minM [1] = (Prod (L (V X)) (L (Neg Y)))
minM [2] = (Prod (L (Neg X)) (L (V Y)))
minM [3] = (Prod (L (Neg X)) (L (Neg Y)))
```

```
minM _ = error "error"
```

Respuesta práctica: Listas de longitud par

```
{-
Módulo      :  EList
Descripción :  Módulo correspondiente a la práctica de
               Listas de longitud par.
Copyright   :  (c) <Daniela Calderón Pérez>
-}

module EList where

--Tipo para las listas de longitud par.
type EList a = [(a,a)]

--Función que regresa la longitud de la lista.
longP :: EList a -> Int
longP [] = 0
longP (x:xs) = 2 + longP xs

--Función que determina si un elemento pertenece a la lista.
elemP :: Eq a => a -> EList a -> Bool
elemP a [] = False
elemP a ((x,y):xs) = if x==a || y==a then True
                     else elemP a xs

--Función que agrega un par al principio de una lista.
consP :: (a,a) -> EList a -> EList a
consP p l = p:l

--Función que agrega un elemento al final de una lista.
snocP :: (a,a) -> EList a -> EList a
snocP x xs = appendP xs (consP x [])

--Función que devuelve el i-ésimo elemento de una lista.
atP :: EList a -> Int -> a
```

```
atP [] n = error "No hay suficientes elementos"
atP (x:xs) 1 = fst x
atP (x:xs) 2 = snd x
atP (x:xs) n = atP xs (n-2)

--Función que convierte una EList a lista primitiva de Haskell.
aplanaP :: EList a -> [a]
aplanaP [] = []
aplanaP ((x,y):xs) = x:y:(aplanaP xs)

--Función que convierte una lista primitiva de Haskell a EList.
toEL :: [a] -> EList a
toEL [] = []
toEL (x:[]) = []
toEL (x:y:xs) = (x,y):(toEL xs)

--Función que elimina los n primeros números de una lista.
dropN :: Int -> EList a -> [a]
dropN 0 xs = aplanaP xs
dropN n [] = []
dropN 1 (x:xs) = (snd x):(aplanaP xs)
dropN n (x:xs) = dropN (n-2) xs

--Función que toma los primeros n elementos de una lista.
--n debe ser par.
takeP :: Int -> EList a -> EList a
takeP 0 xs = []
takeP n [] = []
takeP 1 (x:xs) = error "No es un numero par"
takeP n (x:xs) = x:(takeP (n-2) xs)
```

Respuesta práctica: Multiconjuntos

```
{- |
Módulo      :  Multiconjuntos
Descripción :  Módulo correspondiente a la práctica de
               Multiconjuntos.
Copyright   :  (c) <Daniela Calderón Pérez>
-}

module Multiconjuntos where

--Tipo para el multiconjunto.
type Multi a = [(a,Int)]

--Función que determina si un elemento pertenece al multiconjunto.
mElem :: (Eq a) =>  Multi a -> a -> Bool
mElem [] e = False
mElem (x:xs) e = if e == (fst x) then True else
                 mElem xs e

--Función que regresa la multiplicidad de un elemento.
mult :: (Eq a) => a -> Multi a -> Int
mult a []  = 0
mult a (x:xs) = if (fst x) == a then snd x else
                  mult a xs

--Función que determina la igualdad de los multiconjuntos.
igualM :: (Eq a) => Multi a -> Multi a -> Bool
igualM [] [] = True
igualM [] ys = False
igualM xs [] = False
igualM (x:xs) ys = if (getElem (fst x) ys) == (snd x)
                       then igualM xs (elimina (fst x) ys)
                   else False
```

```
--Función que realiza la interseccion de dos multiconjuntos.
interM :: Eq a => Multi a -> Multi a -> Multi a
interM [] m2 = []
interM m1 [] = []
interM (x:xs) m2 = let n = getElem (fst x) m2 in
                   if n == 0 then interM xs m2
                   else if n >= (snd x) then
                        x:(interM xs (elimina (fst x) m2))
                        else (fst x, n):(interM xs (elimina (fst x) m2))

--Función que realiza la diferencia de dos multiconjuntos.
difM :: Eq a => Multi a -> Multi a -> Multi a
difM [] m2 = []
difM m1 [] = m1
difM (x:xs) m2 = let n = getElem (fst x) m2 in
                 if n >= snd x then difM xs m2
                 else ((fst x), (snd x)-n):(difM xs m2)

--Función que agrega un elemento al multiconjunto.
agregaM :: Eq a => a -> Multi a -> Multi a
agregaM a [] = [(a,1)]
agregaM a (x:xs) = if (fst x) == a then (a,(snd x)+1):xs
                   else x:(agregaM a xs)
```

Respuesta práctica: Conversiones numéricas

```
{-
Módulo      :  Conversiones
Descripción :  Módulo correspondiente a la práctica de
               conversiones numéricas.
Copyright   :  (c) <Daniela Calderón Pérez>
-}

module Conversiones where

--Tipos para las diferentes bases.
type Binario = [Int]
type Octal = [Int]
type Hexa = [String]

--Función que transforma un decimal a binario.
decToBin :: Int -> Binario
decToBin n = decToBin' (abs n) []

--Función auxiliar para decToBin
decToBin' :: Int -> [Int] -> Binario
decToBin' n l = if n < 1 then l
                else let (a,b) = divMod n 2 in
                                 decToBin' a (b:l)

--Función que transforma un binario a decimal.
binToDec :: [Int] -> Int
binToDec xs = binToDec' xs ((length xs) - 1)

--Función auxiliar para binToDec
binToDec' :: [Int] -> Int -> Int
binToDec' [] n = 0
binToDec' (x:xs) n = (x*(2^n)) + binToDec' xs (n-1)
```

```
--Función que transforma un numero octal a decimal.
octToDec :: Octal -> Int
octToDec xs = octToDec' xs ((length xs) - 1)

--Función auxiliar para octToDec
octToDec' :: Octal -> Int -> Int
octToDec' [] n = 0
octToDec' (x:xs) n = (x*(8^n)) + octToDec' xs (n-1)

--Función que transforma un octal a binario.
octToBin :: Octal -> Binario
octToBin [] = []
octToBin (x:xs) = (decToBinR x 3) ++ octToBin xs

--Función auxiliar para octToBin.
decToBinR :: Int -> Int -> Binario
decToBinR n m = let b = decToBin n in
                  rellena m (m - (mod (length b) m)) b

--Función que transforma un decimal a hexadecimal
decToHexa :: Int -> Hexa
decToHexa n = decToHexa' (abs n) []

--Función auxiliar para decToHexa.
decToHexa' :: Int -> [String] -> Hexa
decToHexa' n l = if n < 1 then l
                 else let (a,b) = divMod n 16 in
                           decToHexa' a ([(auxHexa b)] ++ l)

--Función que transforma un binario a hexadecimal.
binToHexa :: Binario -> Hexa
binToHexa xs = binToHexa' $ separa 4
                      $ rellena 4 (4 - (mod (length xs) 4)) xs

--Función auxiliar para binToHexa.
binToHexa' :: [Binario] -> Hexa
binToHexa' [] = []
binToHexa' (x:xs) = (auxHexa (binToDec x)):(binToHexa' xs)

--Función que transforma un hexadecimal a binario.
hexaToBin :: Hexa -> Binario
hexaToBin [] = []
hexaToBin (x:xs) = (decToBinR (auxHexa2 x) 4) ++ hexaToBin xs
```

Respuesta práctica: Permutaciones de listas

```
{-
Módulo      :  Permutaciones
Descripcion :  Módulo correspondiente a la práctica de
               permutaciones.
Copyright   :  (c) <Daniela Calderón Pérez>
-}

module Permutaciones where

--Función que determina si un conjunto es permutacion de otro.
--Solución uno.
esPer1 :: (Ord a ,Eq a) => [a] -> [a] -> Bool
esPer1 [] [] = True
esPer1 [] xs = False
esPer1 xs [] = False
esPer1 (x:xs) ys = if (length xs) /= (length ys) then False
                   else esPer1 xs (elimina x ys)

--Función que elimina un elemento de la lista.
elimina :: Eq a => a -> [a] -> [a]
elimina a [] = []
elimina a (x:xs) = if x == a then  xs
                   else  x:(elimina a xs)

--Función que determina si un conjunto es permutacion de otro.
--Solución tres.
esPer3 :: (Ord a ,Eq a) =>  [a] ->[a] -> Bool
esPer3 xs ys = if (length xs) /= (length ys) then False
               else quickSort xs == quickSort ys

--Función quickSort
quickSort :: (Ord a) => [a] -> [a]
quickSort [] = []
```

```
quickSort (x:xs) =
    let sS = quickSort [a | a <- xs, a <= x]
        bS = quickSort [a | a <- xs, a > x]
    in  sS ++ [x] ++ bS
```

Respuesta práctica: Notación prefija, infija y sufija

```
{-
Módulo      :  Notaciones
Descripción :  Módulo correspondiente a la práctica de
               notación infija, sufija y prefija.
Copyright   :  (c) <Daniela Calderón Pérez>
-}

module Notaciones where

--Importar módulos para realizar operciones específicas de
--Data.String y Data.Char.

import Data.String
import Data.Char

--Tipos para implementar el modulo.
data Op = Suma | Resta | Multi | Div deriving (Show,Eq,Ord)
data Par = L | R deriving (Show,Eq)
data Token = TO Op | TP Par | NumE Int deriving (Show,Eq)

type Prefijo = [Token]
type Infijo = [Token]
type Posfijo = [Token]

--Función que transforma a prefijo.
transformaPr :: String -> Prefijo
transformaPr s = auxTPr (words s) []

--Auxiliar de transformaPr.
auxTPr :: [String] -> [Token] -> Prefijo
auxTPr [] [] = []
auxTPr [] r = r
auxTPr (x:xs) r = if esDigito x
```

```
                then auxTPr xs  (r ++ [NumE (read x ::Int)])
                else auxTPr xs (r ++ [TO (getOp x)])

--Función que regresa un operador.
getOp :: String -> Op
getOp "+" = Suma
getOp "-" = Resta
getOp "*" = Multi
getOp "/" = Div
getOp _ = error "Operador no definido"

--Función que transforma a infijo.
transformaIn :: String -> Infijo
transformaIn s = auxTIn (words s) []

--Auxiliar de transformaIn.
auxTIn :: [String] -> [Token] -> Infijo
auxTIn [] [] = []
auxTIn [] r = r
auxTIn ("(":xs) r = auxTIn xs (r ++ [TP L])
auxTIn (")":xs) r = auxTIn xs (r ++ [TP R])
auxTIn (x:xs) r = if esDigito x
                then auxTIn xs  (r ++ [NumE (read x ::Int)] )
                else auxTIn xs (r ++ [TO (getOp x)])


--Función que evalúa una expresión prefija.
evalPr :: Prefijo -> Int
evalPr s = auxEPr (reverse s) []
```

Respuesta práctica: Enteros y racionales como pares

```
{-
Módulo      :  EntRel
Descripción :  Módulo correspondiente a la práctica de
               enteros y racionales como pares.
Copyright   :  (c) <Daniela Calderón Pérez>
-}

module EntRel where

--Tipos para enteros y racionales.
type Entero = (Int,Int)
type Racional = (Int,Int)


--Función que tranforma un Entero a Int de Haskell.
toInt :: Entero -> Int
toInt (a,b) = a-b

--Función que transforma un Int a Entero.
toEntero :: Int -> Entero
toEntero n = (n,0)

--Función que suma dos Enteros.
sumaE :: Entero -> Entero -> Entero
sumaE a b = ((fst a + fst b), (snd a + snd b))

--Función que resta dos Enteros.
restaE :: Entero -> Entero -> Entero
restaE a b = ((fst a - fst b), (snd a - snd b))

--Función que multiplica dos Enteros.
multiE :: Entero -> Entero -> Entero
multiE (a,b) (c,d) = ((a*c)+(b*d), ((a*d)+(b*c)))
```

```
--Función que transforma un Racional a Float.
toFloat :: Racional -> Float
toFloat (a,b) = (fromIntegral a) / (fromIntegral b)

--Función que suma dos Racionales.
sumaR :: Racional -> Racional -> Racional
sumaR (a,b) (c,d) = (((a*d)+(b*c)),(b*d))

--Función que resta dos Racionales.
restaR :: Racional -> Racional -> Racional
restaR a b = sumaR a (inverso b)

--Función que multiplica dos Racionales.
multiR :: Racional -> Racional -> Racional
multiR (a,b) (c,d) = ((a*c),(b*d))

--Función que divide dos Racionales.
divR :: Racional -> Racional -> Racional
divR (a,b) (c,d) = multiR (a,b) (d,c)
```

Respuesta práctica: Conjuntos como funciones

```
{-
Módulo      :  ConjuntosF
Descripción :  Módulo correspondiente a la práctica de
               conjuntos como funciones.
Copyright   :  (c) <Daniela Calderón Pérez>
-}

module ConjuntosF where

--Tipo para conjuntos como funciones.
type Set = Int -> Bool

--Conjunto que representa a los pares.
pares :: Set
pares n = (mod n 2) == 0

--Función que determina si un elemento pertenece a un conjunto.
elemS :: Int -> Set  -> Bool
elemS x f = f x

--Función que agrega un elemento a un conjunto.
agregar :: Int -> Set -> Set
agregar a f = if f a then f else g
         where g x  = if x == a then True else f x

--Función que realiza la unión de dos conjuntos.
unionC :: Set -> Set -> Set
unionC f g = h
      where h x = f x || g x

--Función que realiza la intersección de dos conjuntos.
inter :: Set -> Set -> Set
inter f g = h
```

```
        where h x = f x && g x

--Función que realiza el complemento de un conjunto.
complemento :: Set -> Set
complemento f = h
              where h x = not (f x)
```

Bibliografía

[1] Miranda Perea, F., Viso Gurovich, E. (2016). Matemáticas Discretas. 2nd ed. Ciudad de México: Las prensas de ciencias.

[2] Miranda Perea, F., González Huesca L. (2019). Lenguajes de programación, Nota de clase 1: Introducción. Ciudad de México, UNAM.

[3] Bravo Mojica, A., Rincón Mejía, H. and Rincón Orta, C. (2006). Álgebra Superior. 1st ed. Ciudad de México: Las prensas de ciencias.

[4] Hutton, G. (2007). Programming in Haskel. Cambridge: Cambridge University Press.

[5] Hall, C., O'Donnell, J. and Page, R. (2006). Discrete mathematics using a computer. London: Springer.

[6] Lauro Aguilar M. (2015). Manufactura de tipos de datos mediante multiconjuntos (Tesis de licenciatura). Universidad Nacional Autónoma de México, Ciudad de México, México.

[7] Aprendehaskell.es. (2019). ¡Aprende Haskell por el bien de todos!. [online] Available at: http://aprendehaskell.es/.

[8] Bird, R. (2000). Introducción a la programación funcional con Haskell. 2da ed. Reino Unido: University of Oxford.