Universidad Nacional Autónoma de México

Facultad de Ciencias

Manual de prácticas para la asignatura de
Lógica Computacional

REPORTE DE ACTIVIDAD DOCENTE

que para obtener el título de:

Licenciado en Ciencias de la Computación

presenta:

Luis Fernando Loyola Cruz

Tutor:

L. en C.C. Manuel Soto Romero

Índice general

II Programación Lógica

III Razonamiento Ecuacional y Deducción Natural

Introducción

Lógica Computacional es una materia obligatoria del cuarto semestre de la Licenciatura en Ciencias de la Computación en la Facultad de Ciencias de la Universidad Nacional Autónoma de México.

Tiene como principal objetivo conocer y aplicar la lógica como una herramienta formal de apoyo en diversas áreas de las Ciencias de la Computación [28].

Objetivos

Puesto que la parte práctica tiene un rol importante en la enseñanza de la materia [28], se elaboró este material con los objetivos de complementar los temas vistos y brindar una serie de ejercicios que sirvan como guía para que los estudiantes puedan poner en práctica conceptos aprendidos en clase.

Para sustentar dichos objetivos, y con el temario de la materia como referencia, se propone el siguiente plan de trabajo:

- Familiarizar al alumno con el paradigma de Programación Declarativa mediante el lenguaje de programación Haskell.
- Revisar desde un enfoque práctico los conceptos de la Lógica Proposicional y la Lógica de Primer Orden, como son: sintaxis, semántica, formas normales, satisfacibilidad, entre otros.
- Implementar algunos de los algoritmos que se proponen en el plan de estudios de la materia siguiendo el enfoque de Programación Funcional.
- Dar un primer acercamiento al paradigma de la Programación Lógica mediante el lenguaje de programación Prolog.

- Hacer uso de algún asistente de pruebas con el fin de profundizar en temas como el Razonamiento Ecuacional y la Deducción Natural, así como dar una introducción a la Verificación Formal.

Estructura del trabajo

En este trabajo se presentan una serie de prácticas y proyectos para la asignatura de Lógica Computacional de la Licenciatura en Ciencias de la Computación, en la Facultad de Ciencias de la UNAM, las cuales tienen como fin guiar al estudiante en el proceso de aprendizaje de los temas vistos en clase desde un punto de vista práctico. Es importante mencionar que este material no sustituye bajo ninguna circunstancia las clases teórico/prácticas del curso.

Se presentan un total de ocho prácticas distribuidas a lo largo de un semestre con duración de 4 meses (16 semanas) y que pueden ser resueltas en periodos de 2 semanas. Además se presentan 3 propuestas de proyectos con el objetivo de aplicar los conocimientos adquiridos a lo largo de cada apartado a la resolución de problemas de mayor interés que pueden aplicarse en otras áreas.

Cada práctica y proyecto está dividido en dos partes: una parte introductoria que brinda un marco teórico y una sección de ejercicios que permiten a los estudiantes poner en práctica los conceptos presentados tanto en clase como en el marco teórico del mismo.

Los temas que cubre este manual, así como las prácticas y proyectos se dividen de la siguiente manera:

Lógica Proposicional y Lógica de Primer Orden (4 prácticas, 1 proyecto)

En este apartado se presentan 4 prácticas: la primera, con ejercicios que le permitirán al alumno familiarizarse con conceptos básicos del lenguaje de programación Haskell, la segunda y tercera que sirven como introducción a la sintaxis y semántica de la Lógica Proposicional y la cuarta que plantea una serie de ejercicios donde se abordan aspectos tanto sintácticos como semánticos de la Lógica de Primer Orden.

El proyecto propuesto en esta sección tiene como objetivo revisar e implementar el algoritmo Davis-Putnam-Logemann-Loveland (DPLL) con el fin de profundizar en algunos temas como Formas Normales, Resolucion Binaria, satisfacibilidad y solucionadores SAT.

Programación Lógica (2 prácticas, 1 proyecto)

En este apartado se presentan dos prácticas: la primera, de introducción al lenguaje de programación PROLOG, contiene ejercicios que le permitirán al alumno familiarizarse con los aspectos sintácticos y semánticos del lenguaje para poder resolver problemas dentro del Paradigma Lógico.

La segunda práctica, aborda conceptos avanzados del lenguaje con el fin de que el alumno los relacione con los mecanismos de control de programas de PROLOG presentes en la búsqueda de soluciones tales como el mecanismo de retroceso, el operador de corte y la negación como falla.
El proyecto de este apartado es una propuesta donde se pueden aplicar los conceptos de la programación lógica a la resolución del problema de generar horarios escolares.

Razonamiento Ecuacional y Deducción Natural (2 prácticas, 1 proyecto)

En esta última sección se presentan dos prácticas: la primera, que pretende familiarizar al alumno con el asistente de pruebas COQ, mientras que la segunda aborda conceptos avanzados con el fin de poder realizar demostraciones haciendo uso del asistente por medio de inducción.

El proyecto correspondiente en esta sección tiene como finalidad aplicar los conceptos vistos en las prácticas así como dar una breve introducción a la Verificación Formal de Programas.

Dado que muchas de las definiciones y algoritmos que se ven a lo largo del curso son recursivos, este trabajo sigue un enfoque declarativo, por lo que para las primeras 4 prácticas se

sigue un estilo funcional por medio del lenguaje de programación Haskell[1]. .

A partir de la quinta práctica, se introduce el estilo de Programación Lógica usando el lenguaje de programación Prolog. Por último, para la sexta y séptima práctica se hace uso del asistente de pruebas Coq[2].

Con este trabajo también se pretende reforzar el dominio de algunos temas correspondientes a temarios de otros cursos obligatorios u optativos como Lenguajes de Programación, Programación Declarativa, Semántica y Verificación, Razonamiento Automatizado, entre otros.

[1]Aunque el material de estas secciones puede ser cubierto con cualquier otro lenguaje de programación se decidió hacer uso de Haskell por su flexibilidad al abstraer y formalizar nociones matemáticas como las presentadas en este material, todo esto gracias a que sigue un estilo de Programación Funcional.

[2]No obstante el material de esta sección puede ser cubierto con algún otro asistente de pruebas como Isabelle ó Agda.

Parte I

Lógica Proposicional y Lógica de Primer Orden

Práctica 1

Introducción a Haskell

1.1. Objetivos

- ▷ Familiarizar al alumno con el lenguaje de programación Haskell, así como con el intérprete `GHCi`, el cual se usa para desarrollar ésta y otras prácticas.

1.2. Introducción

En esta práctica revisaremos de manera breve e informal los conceptos básicos del lenguaje de programación Haskell con el fin de que el alumno pueda escribir programas en el mismo.[1]

1.2.1. Haskell

A lo largo del curso se trabaja con conceptos matemáticos pertenecientes a la Lógica, en particular, la Lógica Computacional. Es por ello que en estos primeros capítulos se usa el lenguaje de programación Haskell.

Uno de los principales motivos por los que se propone el uso de Haskell es la *transparencia referencial*[2], que es una propiedad de algunos lenguajes de programación funcionales que establece que se puede reemplazar una expresión con otra de igual valor en cualquier lugar

[1] Estos conceptos ser abordan de manera rápida e informal, por lo que si requiere y/o desea ahondar más en Haskell puede consultar los recursos listados en las referencias [7, 14, 18].

[2] A los lenguajes que tienen esta propiedad se les denomina *puros*.

de un programa sin cambiar el significado de éste [22]. Como consecuencia se tiene que una función siempre producirá el mismo resultado para una entrada dada. Por tanto es posible construir, razonar y manipular programas funcionales como si fuesen una expresión matemática. Esto es de gran utilidad para este trabajo, ya que muchas funciones y algoritmos que se estudian en el curso se pueden definir de manera casi directa en HASKELL.

1.2.2. Instalación de GHC (*Glasgow Haskell Compiler*)

Antes de comenzar a escribir programas en HASKELL se necesita instalar un compilador. `GHC` *(Glasgow Haskell Compiler)* es el compilador más usado hoy en día para HASKELL, y el cual se utiliza para la primera parte de este manual.

Para instalar `GHC` así como otras herramientas que pueden ser de utilidad, visitar https://www.haskell.org/platform/ y dependiendo del sistema operativo que se esté usando seguir las instrucciones correspondientes.

Por ejemplo, para instalar la plataforma de HASKELL en una distribución de Linux basada en Debian se usa el siguiente comando:

```
> sudo apt-get install haskell-platform
```

O si solo desea instalar `GHC` basta con ejecutar el comando:

```
> sudo apt-get install ghc
```

Se puede verificar la instalación ejecutando en terminal el comando `ghc -version` y esperar como resultado la versión del compilador.

```
> ghci --version
The Glorious Glasgow Haskell Compilation System, version 9.0.1
```

`GHC` incluye un ambiente interactivo que lleva por nombre `GHCi`[3]. Con `GHCi` se pueden evaluar de manera interactiva expresiones escritas en HASKELL e interpretar programas. Para usar éste ambiente interactivo basta con ejecutar el comando `ghci`.

En este manual se usa principalmente `GHCi` debido a su naturaleza interactiva, que lo hace idóneo para propósitos educativos. No obstante, si se requiere un mayor rendimiento o un binario ejecutable resultado de un programa escrito en HASKELL se recomienda hacer uso del compilador.

Todos los comandos que se usan dentro del ambiente interactivo `GHCi` comienzan con `:` seguido del nombre del comando y una serie de cero o más argumentos. Entre los comandos más útiles se tienen:

- `:load <archivo>`, que sirve para cargar las definiciones de un archivo y poder usarlas en el ambiente. Su abreviación es `:l <archivo>`. Por convención la extensión de los archivos que contengan definiciones escritas en HASKELL es `hs`.
- `:reload` intenta actualizar los módulos previamente cargados, o cualquier módulo dependiente, si alguno ha cambiado. También se puede usar `:r`
- `:quit`, para salir del ambiente interactivo. Su abreviación es `:q`
- `:help`, para obtener más información sobre los comandos disponibles y su uso. También es posible usar `:?`

1.2.3. Tipos primitivos

HASKELL provee una serie de tipos primitivos, es decir, tipos de datos que ya vienen predefinidos en HASKELL.

Los más comunes son:

[3]Para más información sobre `GHCi` se recomienda revisar la documentación oficial.

Tipo	Ejemplo
`Int`[4]	...,-2, -1, 0, 1, 2, ...
`Float`	`1.0`, `-1.25`, `2.5`, etc.
`Bool`	`True`, `False`
`Char`	‘a’, ‘b’, ‘c’, ’\n’, etc.
`String`[5]	“hola mundo”, “abc”, etc.

Si bien, HASKELL nos da la flexibilidad de escribir expresiones sin especificar el tipo de éstas, es una buena práctica de programación poner el tipo explicito. Para indicar que una expresión `e` tiene el tipo `t` se escribe `e :: t`.

```
ghci> "haskell" :: String
"haskell"
ghci> True :: Bool
True
ghci> 1 :: Int
1
```

[4]No confundir con el tipo de dato `Integer`. `Int` es un tipo de dato primitivo que corresponde a un entero de 32 bits, mientras que `Integer` es un tipo de dato numérico de precisión arbitraria, lo que significa que no tiene límites en cuanto al tamaño de los valores que puede representar.

[5]Nótese la diferencia en el uso de las dobles comillas para el tipo `String` en vez de comillas simples, que se usan para el tipo `Char`.

Tip

En `GHCi` se puede conocer el tipo de una expresión usando el comando `:type <expresión>`, o su forma abreviada `:t <expresión>`.

```
ghci> :t "hola mundo"
"hola mundo" :: String
```

1.2.4. Funciones

Dado que HASKELL es un lenguaje de programación funcional, es de esperar que las funciones jueguen un papel esencial.

En matemáticas, una función es un relación de elementos de un conjunto A a elementos de un conjunto B, en donde un elemento de A es mapeado a un único elemento en B. Por ejemplo, la función que eleva un número al cuadrado toma un elemento x del conjunto de los números enteros y mapea x a un elemento del mismo conjunto.

$$f : \mathbb{Z} \to \mathbb{Z}$$
$$f(x) = x^2$$

Para evaluar la función simplemente se sustituyen los parámetros de la función (en este caso x) por un valor.

$$f(3) = 3^2 = 9$$

En HASKELL la definición de ésta función[6] es la siguiente:

```
f :: Int -> Int
f x = x^2
```

Código 1.1: Función que calcula el cuadrado de un número

[6]Los nombres de las funciones deben empezar con letras minúsculas.

En la Sección 1.2.3 se introdujo el símbolo `::`, que sirve para indicar que una expresión tiene un cierto tipo. La primera línea corresponde a la declaración de tipo de la función. Si `t1` y `t2` son tipos, entonces `t1` $\rightarrow$ `t2` es el tipo de una función que toma un elemento de tipo t1 y devuelve un resultado de tipo `t2`. En términos de dominio y codominio, el dominio de la función es el tipo `t1` y el codominio es el tipo `t2`.

La segunda línea corresponde a la declaración de la función. Nótese que la manera en como se define una función puede verse como una ecuación en el sentido matemático, pues se indica que el lado izquierdo y el lado derecho denotan el mismo valor. Estas ecuaciones son de la forma `nombre` $p_1, p_2, \ldots, p_n$ `=` `cuerpo` donde *nombre* es el nombre de la función, $p_1, p_2, \ldots, p_n$ son los parámetros[7]de la función y *cuerpo* es una expresión que define el resultado de la función.

Un programa en HASKELL es entonces el resultado de definir y aplicar funciones. Al igual que en matemáticas, aplicar una función consiste en reemplazar los parámetros de una función por valores concretos y evaluar el cuerpo.

```
ghci> f 3
9
```

Las funciones en HASKELL, así como en matemáticas pueden ser multi-paramétricas[8]. Considérese la función `suma`, que como argumentos recibe dos números y como resultado se obtiene otro número. El tipo de la función se ve de la siguiente forma:

```
suma :: Int -> Int -> Int
```

[7]Es importante hacer la distinción entre parámetros y argumentos. Los parámetros son los nombres que se usan en la definición de la función, mientras que los argumentos son los valores que se pasan a la función al momento de evaluarla.

[8]En HASKELL esto se debe a la *currificación*, que es el proceso de transformar una función que toma múltiples parámetros, en una función que toma solo un parámetro y devuelve otra función que acepta los parámetros restantes [7].

Básicamente si se quiere definir una función que reciba n parámetros, el tipo de la función se ve de la siguiente manera:

$$\texttt{f :: } t_1 \texttt{ -> } t_2 \texttt{ -> } \ldots \texttt{ -> } t_n \texttt{ -> b}$$

siendo t_i el tipo del i-ésimo parámetro y `b` el tipo del valor a devolver.

Así como hay tipos predefinidos en HASKELL, también hay funciones predefinidas. Entre las más comunes estan:

Funciones aritméticas

- `(+)`: Recibe dos números y regresa la suma de éstos.

```
ghci> 1 + 2
3
```

- `(-)`: Regresa la resta de su segundo argumento con el primero.

```
ghci> 2 - 2
0
```

- `(*)`: Computa la multiplicación de dos números.

```
ghci> 5 * 5
25
```

- `div`: Computa la división entera entre dos números.

```
ghci> div 6 2
3
```

- `(/)`: Computa la división real entre dos números.

```
ghci> 5 / 2
2.5
```

- `(^)`: Devuelve el resultado de elevar su primer argumento al segundo.

```
ghci> 2^3
8
```

Funciones booleanas

- `not`: Regresa la negación lógica del argumento que recibe.

```
ghci> not True
False
```

- `(&&)`: Regresa la conjunción lógica de sus dos argumentos.

```
ghci> True && False
False
```

- `(||)`: Regresa la disyunción lógica de sus dos argumentos.

```
ghci> True || True
True
```

Funciones de comparación

- `(==)`: Devuelve `True` si los dos argumentos que recibe son iguales. `False` en otro caso.

```
ghci> "haskell" == "haskell"
True
```

- `(/=)`: Devuelve `True` si los dos argumentos que recibe son distintos entre si. `False` en otro caso.

```
ghci> 10 /= 100
True
```

- `(<)`: Devuelve `True` si el primer argumentos es estrictamente menor que el segundo. `False` en otro caso.

```
ghci> 0 < 1
True
```

- `(<=)`: Devuelve `True` si el primer argumento es menor o igual al segundo. `False` en otro caso.

```
ghci> 1 <= 1
True
```

- `(>)`: Devuelve `True` si el primer argumento es estrictamente mayor que el segundo. `False` en otro caso.

```
ghci> 2 > 1
True
```

- `(>=)`: Devuelve `True` si el primer argumento es mayor o igual al segundo. `False` en otro caso.

```
ghci> 10 >= 9
True
```

Tip

Existe una herramienta llamada Hoogle que es un buscador de funciones de Haskell. Se pueden buscar funciones, ya sea por nombre o por el tipo de la función.

1.2.5. Operadores y precedencia

Nótese como las funciones aritméticas, las funciones booleanas, las de comparación, entre otras, se usan de manera infija. Esto se debe a que son definidas como operadores.

En HASKELL los nombres de las funciones deben iniciar con una letra minúscula seguida de una serie de caractéres alfanuméricos. Sin embargo, es posible definir funciones, conocidas como operadores, cuyo nombre este formado por una serie de cualquier combinación de los siguientes símbolos, delimitados por paréntesis [21] :

```
! # $ * + . / < = > ? \ ^ | : - ~ %
```

La principal diferencia entre los operadores y las funciones es que los operadores se usan de manera infija, mientras que las funciones se usan de manera prefija. Por ejemplo, tómese en cuenta la función presentada en el Código 1.2, la cual recibe dos números y regresa la suma de sus cuadrados.

```
suma_cuadrados :: Int -> Int -> Int
suma_cuadrados x y = x^2 + y^2
...

ghci> suma_cuadrados 2 3
13
```

Código 1.2: Función que calcula la suma de los cuadrados de dos números

Si se quiere usar esta función de manera infija entonces se delimita el nombre de ésta entre el símbolo que representa al acento grave (`).

```
ghci> 2 `suma_cuadrados` 3
13
```

Sin embargo, por motivos de legibilidad es conveniente definir un operador, como se muestra en el Código 1.3.

```
(<+<) :: Int -> Int -> Int
(<+<) x y = x^2 + y^2
...

ghci> 2 <+< 3
13
```

Código 1.3: Operador que calcula la suma de los cuadrados de dos números

Nota

Si se desea usar el operador de manera prefija, basta con hacer escribir el operador entre paréntesis, como en la definición de éste.

```
ghci> (<+<) 2 3
13
```

Dado que al usar la notación infija se elimina el uso de paréntesis, es necesario especificar la precedencia y asociatividad al definir operadores. Para indicar que un operador asocia a la izquierda, se usa la sentencia `infixl n` $op_1, op_2, \ldots, op_n$. De manera similar ocurre con los operadores que asocian a la derecha, para los cuales se usa `infixr` y si es el caso en que el operador es no asociativo se usa `infix`. La n es un número que puede ir del 0 al 9 y según éste, será la precedencia del operador. Un número mayor indica que el operador tiene mayor precedencia frente a los operadores con un número menor. El segundo argumento es una lista de operadores a los cuales se les aplicará la precedencia y/o asociatividad definida [21].

Nota

Por defecto todos los operadores son asociativos a la izquierda y tienen una precedencia de 9 [21].

Por ejemplo, la asociatividad y precedencia de las operaciones aritméticas básicas se muestra en el Código 1.4.

```
infixl 6 +, -
infixl 7 *, /
```

Código 1.4: Precedencia y asociatividad de los operadores aritméticos

Si se quisiera que el operador <+< tuviera menor precedencia que la suma y resta, entonces se debería definir como en el Código 1.5.

```
infixl 7 <+<
...

ghci> 2 <+< 3 - 1
8
```

Código 1.5: Precedencia y asociatividad del operador <+<

1.2.6. Funciones anónimas

Las funciones anónimas[9], como su nombre indica, son funciones que no tienen un nombre y por lo tanto no pueden ser utilizadas fuera del alcance en dónde fueron declaradas.

Las funciones anónimas generalmente son usadas en un contexto donde el uso de éstas es único, por ejemplo, cuando se hace uso de funciones de order superior, de las cuales se habla en la Sección 1.2.15.

La sintaxis para declarar funciones anónimas es la siguiente:

[9]También conocidas como *abstracciones lambda* pues la sintaxis se inspira del Cálculo Lambda, en donde las funciones son de la forma $\lambda x \mapsto e$.

$$\backslash a_1\ a_2\ \ldots\ a_n\ \texttt{-> cuerpo}$$

donde a_i representa el í-esimo parámetro de la función.

Por ejemplo, usando una función anónima se puede definir la suma dos números:

```
ghci> (\ x y -> x+y) 1 2
3
```

1.2.7. Secciones

Las secciones son una forma de definir funciones parciales. Una sección es una expresión que contiene un operador y un número de argumentos parcialmente aplicados. Por ejemplo, la sección `(+1)` es una función que recibe un número y regresa la suma de éste con 1.

Nota

Algunas secciones son equivalentes a funciones anónimas, por ejemplo, la sección `(+1)` es equivalente a la función anónima `(\x -> x+1)`.

Las secciones son útiles porque permiten crear funciones más específicas y expresivas de una manera concisa. Por ejemplo, si se desea calcular la lista de los cuadrados de los primeros 5 números enteros, se podría escribir como sigue:

```
ghci> map (^2) [1,2,3,4,5]
[1,4,9,16,25]
```

En este caso, `(^2)` es una sección de la función potencia `(^2)` que eleva su primer argumento al cuadrado.

Las secciones también son útiles cuando se quieren pasar funciones como argumentos a otras funciones. Por ejemplo, si se desea filtrar una lista de números para obtener sólo los que son mayores que 5, se podría escribir:

```
ghci> filter (>5) [1,2,3,4,5,6,7,8,9,10]
[6,7,8,9,10]
[2,4]
```

En este caso, `(>5)` es una sección de la función de comparación `(>)` que compara su argumento con 5.

1.2.8. Listas

Las listas son una estructura de datos ampliamente usada en Haskell. Una lista es un secuencia de valores del mismo tipo. Para denotar que algo es una lista de elementos de tipo `t` se usa la notación `[t]`.

Una lista se define recursivamente como:

- La lista vacía es una lista.
- Si x[10] es un elemento de tipo A y xs[11] es una lista de elementos de tipo A entonces `cons(x, xs)` es una lista de elementos de tipo A.
- Son todas.

En Haskell la definición de las listas se muestra en el Código 1.6, en donde la lista vacía se denota con `[]`, y el operador *cons*[12], denotado como `(:)`, se usa para construir listas.

```
data [] a = [] | a : [a]
```

Código 1.6: Definición de listas en Haskell

Usando la definición antes mencionada es como se pueden definir listas en Haskell. Por ejemplo, la lista que contiene a los números 1, 2 y 3 en este orden se define como en el Código 1.7:

[10] Usualmente conocido como *cabeza*.
[11] Usualmente conocida como *resto*.
[12] Abreviatura de *construct* (construir en español).

```
lista1 :: [Int]
lista1 = 1:(2:(3:[]))
```

Código 1.7: Definición de una lista usando el operador *cons*

Dado que el operador *cons* asocia a la derecha, la lista del Código 1.7 se puede reescribir eliminando los paréntesis.

Se aprecia que la notación para definir listas usando el operador *cons* es compleja de leer y tediosa de escribir. Para facilitar la definición y lectura de listas, HASKELL añade azúcar sintáctica[13] en la definición de éstas, de forma que se pueden escribir listas como secuencias separadas por comas encerradas entre corchetes.

```
lista2 :: [Int]
lista2 = [1,2,3]
```

Código 1.8: Definición de una lista usando azúcar sintactica

Hay varias funciones de utilidad[14] sobre listas contenidas en el *Preludio*[15] de HASKELL como:

- `head`: Regresa el primer elemento de una lista

  ```
  ghci> head [1,2,3]
  1
  ```

- `last`: Regresa el último elemento de una lista

  ```
  ghci> last [1,2,3]
  3
  ```

- `tail`: Regresa la lista sin el primer elemento

[13] Sintaxis dentro de un lenguaje de programación que facilita la legibilidad o expresibilidad [29].

[14] El módulo `Data.List` define más funciones sobre listas

[15] Módulo estándar que se carga automáticamente al inicio de cualquier programa. Contiene una serie de definiciones de funciones y tipos de datos.

```
ghci> tail [1,2,3]
[2,3]
```

- `length`: Regresa el número de elementos de una lista

```
ghci> length [1,2,3]
3
```

- `null`: Verifica si la lista es vacía o no

```
ghci> null []
True
```

- `elem`: Dado un elemento `e`, verifica si `e` se encuentra en una lista

```
ghci> elem 2 [1,2,3]
True
```

- `(++)`: Dadas dos listas, regresa la concatenación de éstas

```
ghci> [1,2,3] ++ [4,5]
[1,2,3,4,5]
```

- `reverse`: Regresa la lista dada en orden inverso

```
ghci> reverse [1,2,3,4,5]
[5,4,3,2,1]
```

1.2.9. Tuplas

Por otro lado, están las tuplas, que a diferencia de las listas pueden contener elementos de distintos tipos. El tipo de una tupla se denota mediante una secuencia de tipos encerrada entre paréntesis, en donde cada tipo es separado por una `,`. Así, el tipo de una tupla de n elementos se ve de la siguiente forma:

$$(t_1, t_2, \ldots, t_n)$$

Al número de elementos de una tupla se le conoce como *aridad*[16].

A diferencia de las listas, cuyo tipo no refleja información sobre la longitud de éstas, las tuplas si lo hacen.

La sintaxis para definir tuplas es la misma que para denotar el tipo de ellas.

```
tupla1 :: (Int, Bool)
tupla1 =  (1, True)
```

Código 1.9: Ejemplo de creación de una tupla

Las tuplas de aridad dos, son llamadas *pares*. Para este tipo de tupla en particular HASKELL define dos funciones[17].

- `fst`: Dado un par regresa el primer elemento de éste.

  ```
  ghci> fst (1,2)
  1
  ```

- `snd`: Dado un par regresa el segundo elemento de éste.

[16] Del inglés *arity*. Dado que tal palabra no existe en el diccionario de la lengua española, trataremos de evitar su uso.

[17] Usualmente estas funciones también son conocidas como *proyecciones*.

```
ghci> snd (1,2)
2
```

1.2.10. Listas por comprensión

En Teoría de Conjuntos existen varias formas para denotar un conjunto. Entre ellas existe una llamada *por comprensión* que sirve para describir un conjunto indicando las propiedades que deben cumplir los elementos de éste [31].

Usando esta notación se puede definir, por ejemplo, el conjunto de todos los números naturales que son pares.

$$\{k \mid k \text{ es par}\}$$

En Haskell esta idea se aplica de manera similar a las listas. Para definir la lista que contiene los elementos del ejemplo anterior se hace de la siguiente forma:

```
ghci> [ x | x <- [1,2..], x `mod` 2 == 0 ]
```

Nota

La expresión `[1,2..]` es una lista pseudoinfinita[18] que contiene a todos los números enteros positivos y fue definida mediante una *enumeración*, que es una manera de expresar secuencias aritmeticas de manera compacta en Haskell.

Para más información revisar la sección 4.2 de la referencia [3].

[18] Dada la evaluación perezosa en Haskell, solo se evaluaran los elementos de la lista que sean necesarios para la ejecución del programa, lo que significa que no se ocupará una cantidad infinita de memoria. En este sentido, se dice que lista es pseudoinfinita, ya que solo se generan y evalúan los elementos necesarios para la ejecución del programa.

En general una lista por comprensión se ve de la siguiente manera:

$$\texttt{[expr} \mid q_1, q_2, \ldots, q_n\texttt{]}$$

En donde *expr* denota el patrón de los elementos que tendrá la lista resultante, mientras que cada q_i es llamado *cualificador* y denotan propiedades sobre los elementos de la lista.

Existen tres categorías de cualificadores [21]:

- *Generadores*: Son expresiones de la forma `e <- xs`, donde `e` es una variable y `xs` una lista. El generador extrae los elementos de `xs` en el orden en que aparecen.
- *Filtros*: Son expresiones boleanas que sirven para determinar si un elemento pertenecerá o no a la lista. Solo los elementos cuyo predicado sea verdadero formarán parte de la lista.
- *Definiciones locales*: Son expresiones de la forma `let var = exp` y sirven para crear nuevas definiciones que pueden ser usadas en *expr* o en cualificadores subsecuentes.

Las listas por comprensión pueden tener multiples cualificadores de distintas categorías, por ejemplo, si se quiere definir una función que elimine los espacios en blanco de una cadena el Código 1.10 propone una solución usando listas por comprensión.

```
remueveEspacios :: String -> String
remueveEspacios s = [c | c <- s, c /= ' ']
```

Código 1.10: Función que quita los espacios de una cadena

Por otro lado es posible tener multiples generadores, por ejemplo, para generar el producto cartesiano de dos listas, como se hace en el Código 1.11.

```
productoCartesiano :: [a] -> [b] -> [(a,b)]
productoCartesiano xs ys = [(x,y) | x <- xs, y <- ys]
```

Código 1.11: Función que computa el producto cartesiano de dos listas

Es importante mencionar que el orden de los cualificadores afecta el orden de los elementos en la lista resultante. Por ejemplo, si en la lista del Código 1.11 se intercambian los generadores, la lista resultante será la misma, pero en orden inverso.

```
ghci> let xs = [1,2,3]
ghci> let ys = [4,5,6]
ghci > [(x,y) | x <- xs, y <- ys]
[(1,4),(1,5),(1,6),(2,4),(2,5),(2,6),(3,4),(3,5),(3,6)]
ghci > [(x,y) | y <- ys, x <- xs]
[(3,6),(3,5),(3,4),(2,6),(2,5),(2,4),(1,6),(1,5),(1,4)]
```

1.2.11. Condicionales

En matemáticas se pueden definir funciones por partes, esto es, funciones cuya expresión analítica depende de los argumentos de éstas. Por ejemplo, la función *valor absoluto* se puede definir de la siguiente manera:

$$abs(x) = \begin{cases} x & x \geq 0 \\ -x & x < 0 \end{cases}$$

En HASKELL hay varias formas de definir funciones que puedan elegir entre multiples resultados. La más simple de éstas es usar expresiones *if-then-else*, cuya sintaxis es:

$$\text{if } \textit{condicion} \text{ then } \textit{expr}_1 \text{ else } \textit{expr}_2$$

en donde *condicion* debe ser una expresión cuya evaluación dé como resultado un valor booleano, de modo que si el resultado de ésta es `True` el resultado de toda la expresión será $expr_1$, y $expr_2$ en cualquier otro caso. Es importante destacar que tanto $expr_1$ como $expr_2$ deben tener el mismo tipo.

Por tanto, usando esta forma de expresiones se puede definir la función `abs` como se muestra en el Código 1.12.

```
abs :: Int -> Int
abs n = if n < 0 then -n else n
```

Código 1.12: Definición de la función valor absoluto

Nótese que al ser una expresión, el condicional *if-then-else* permite anidar expresiones de esta forma, como se ve en el Código 1.13.

```
sgn :: Int -> Int
sgn n = if n < 0 then -1
            else if n == 0 then 0
                else 1
```

Código 1.13: Definición de la función signo

El Código 1.13 funciona, sin embargo si se tuviera una función con más de 2 casos, definirla usando expresiones *if-then-else* podría ser poco legible. Para resolver este problema HASKELL define un mecanismo conocido como *guardias*.

La sintaxis de las guardias es la siguiente:

$$
\begin{array}{ll}
\texttt{funcion argumentos} & \texttt{| cond}_1\texttt{ = expr}_1 \\
& \texttt{| cond}_2\texttt{ = expr}_2 \\
& \ldots \\
& \texttt{| otherwise = expr}_n
\end{array}
$$

en donde cada $cond_i$ es una expresión booleana. Si la primera condición se evalua a `True` entonces $expr_1$ es el resultado. De otra forma se procede a evaluar $cond_2$ y si la condición se cumple entonces el resultado es $expr_2$ y así sucesivamente. Si ninguna de las condiciones se cumple, entonces el resultado es $expr_n$.

¡Importante!

El orden en que se definen las guardias es importante, ya que si una condición se cumple, el resultado será el de la primera guardia que se cumpla.

¡Importante!

Cada guardia define una ecuación. Por tanto no se debe poner el símbolo `=` entre la guardia y los argumentos de la función.

La palabra reservada `otherwise` es una forma de especificar una condición verdadera[19], y aunque no es necesario terminar la secuencia de guardias con ella, es conveniente ya que ayuda a manejar los casos restantes y por tanto hace que la función sea total[20].

Nota

Las guardias no son expresiones. Por lo que sólo pueden usarse en la definición de una función.

Usando guardias se puede redefinir la función del Código 1.13 como se muestra en el Código 1.14

```
sgn :: Int -> Int
sgn n | n < 0  = -1
      | n == 0 = 0
      | n > 0  = 1
```

Código 1.14: Definición de la función signo usando guardias

1.2.12. Declaraciones locales

En los lenguajes de programación imperativos las variables se usan para hacer referencia a un valor que puede cambiar en tiempo de ejecución. Sin embargo, en Haskell las variables son en realidad identificadores que se asocian con una expresión, y una vez asociadas, el valor de la expresión no cambia. Es decir, en Haskell una variable se asocia con un valor inmutable.

En lugar de modificar el valor de una variable, en Haskell se crean nuevas variables para representar un valor diferente. Esto se logra mediante la expresión `let`, que permite definir una variable y su valor asociado en un contexto local. Esto puede ser de utilidad ya sea para reutilizar el resultado de la evaluación de alguna expresión, o para mejorar la legibilidad del código.

[19]En realidad es azúcar sintáctica para `True`.

[20]Una función total es aquella que está definida para todos los valores de su dominio.

Considera la función que se muestra en el Código 1.15 que calcula las raíces de una ecuación de segundo grado.

```
raices :: Float -> Float -> Float -> (Float, Float)
raices a b c = ((-b + sqrt (b^2 - 4*a*c)) / (2*a) , (-b - sqrt (b^2 -
   4*a*c)) / (2*a))
```

Código 1.15: Función que calcula las raíces de una ecuación de segundo grado

Para mejorar la legibilidad de ésta y evitar repetir la expresión $b^2 - 4 * a * c$ se puede usar una expresión `let` cuya sintaxis es:

```
let decls in expr
```

en donde *decls* es una lista de declaraciones locales de la forma $p_1 = e_1; p_2 = e_2; \ldots; p_n = e_n$ y cada una de éstas puede ser usada dentro de *expr*.

Así, usando expresiones `let` se puede redefinir la función del Código 1.15 como se ve en el Código 1.16.

```
raices :: Float -> Float -> Float -> (Float, Float)
raices a b c = let d = sqrt (b^2 - 4*a*c)
               in ((-b + d) / (2*a) , (-b - d) / (2*a))
```

Código 1.16: Función que calcula las raíces de una ecuación de segundo grado usando `let`

Como se ha podido observar hasta ahora, HASKELL provee diferentes maneras de realizar la misma tarea, cada una con ventajas y desventajas y con sus respectivas diferencias entre sí. Además de las expresiones `let`, existe otra forma de definir variables locales en una función, utilizando la palabra clave `where`.

La sintaxis de la claúsula `where` es la siguiente:

```
funcion parametros = expr
                where declaraciones
```

Mientras que cuando se usa `let` primero se hacen las declaraciones y luego se usan en una expresión, cuando se usa `where` se hace de manera contraria; primero se usan los identificadores

y luego se definen.

```
raices :: Float -> Float -> Float -> (Float, Float)
raices a b c = (x1, x2)
               where d = sqrt (b^2 - 4*a*c)
                     x1 = (-b + d) / (2*a)
                     x2 = (-b - d) / (2*a)
```

Código 1.17: Función que calcula las raices de una ecuación de segundo grado usando `where`

No obstante, la sintaxis no es la única diferencia. La claúsula `where` es más flexible en cuanto al alcance de sus declaraciones. Cuando se usan guardias, se puede usar `where` para declarar identificadores que pueden ser usados en cada una de las guardias, mientras que con `let` el alcance de los identificadores esta restringido por el lado derecho de la ecuación.

```
imc :: Float -> Float -> String
imc peso altura
    | resultado <= 18.5 = "Bajo peso"
    | resultado <= 24.9 = "Peso normal"
    | resultado <= 29.9 = "Sobrepeso"
    | otherwise = "Obesidad"
    where resultado = peso / (altura * altura)
```

Código 1.18: Función que calcula el índice de masa corporal

La función del Código 1.18 toma dos argumentos, peso y altura, y calcula el IMC de una persona. Luego, se usan guardias para evaluar el resultado del cálculo y devolver una cadena de texto que describe el estado del peso de la persona. La variable `resultado` se define en la cláusula `where`, lo que permite calcular el IMC una sola vez y luego reutilizarlo en las diferentes guardias. Esto evita repetir el cálculo de la división `peso / (altura * altura)` en cada guardia.

Tip

También es posible declarar funciones usando expresiones `let` y claúsulas `where`.

```
ghci> let suma a b = a + b in suma 1 2
3
```

```
g :: Int -> Int
g x = f + h
    where f = x * 2
          h = x + 10
```

1.2.13. Recursion

En Haskell, la definición de un cálculo se realiza declarando lo *qué* se desea obtener, a diferencia de los lenguajes de programación imperativos que se enfocan en *cómo* obtener algo. Por esta razón, en Haskell no se utilizan estructuras de repetición como for o while, sino que se hace uso de la recursión.

La recursion es el proceso de resolver o definir un problema en términos de si mismo[21]. Una función recursiva es entonces una función que se llama a sí misma en su definición. Sin embargo para que una definición recursiva tenga sentido, se tiene que eventualmente llegar a un caso base. Por tanto podemos decir que una función recursiva consta de dos partes:

- *Caso base*[22]: Establece una condición de término en donde la función regresa un resultado concreto. Usualmente es la solución de la versión más simple de un problema.
- *Caso recursivo*: Se trata de resolver el problema inicial resolviendo una versión más simple del mismo. Es aquí dónde la función se llama a sí misma, pero, usualmente, con una entrada más pequeña, para eventualmente llegar al caso base.

En matemáticas existe una función, ampliamente usada en el área de combinatoria, llamada

[21] En un principio se usan versiones más pequeñas del problema, aunque no siempre ocurre de esa manera.
[22] Algunos autores le llaman *Cláusula de escape*.

factorial, que recibe como argumento un número entero positivo n y como resultado se obtiene la multiplicación de todos los números enteros positivos menores o iguales a n. Una manera de definir la función factorial puede ser usando el operador producto.

$$fact(n) = \prod_{i=1}^{i=n} i$$

Nótese como la definición anterior nos indica como es que se calcula el factorial. En un lenguaje imperativo, como JAVA, la implementación es directa haciendo uso de alguna estructura de repetición como se observa en el Código 1.19.

```
public int factorial(int n){
    fact = 1;
    for(int i = 1; i <= n; i++)
        fact *= i;
    return fact;
}
```

Código 1.19: Implementación de la función *factorial* en Java

A modo de ejemplo, considérese el factorial de los primeros cuatro números positivos.

$$fact(4) = 4 \times 3 \times 2 \times 1$$
$$fact(3) = 3 \times 2 \times 1$$
$$fact(2) = 2 \times 1$$
$$fact(1) = 1$$

Se puede notar como el factorial de 3 esta incluido en el factorial de 4, y en general el factorial de cualquier número n es el resultado de multiplicar n por el factorial de $n-1$, aunque hay una excepción: el cero. Si el factorial de cero se calculase de la misma forma que el resto de los números entonces el resultado sería la multiplicación del número 0 por el factorial de -1, sin embargo la función factorial solo ésta definida sobre los números enteros positivos. Por tanto, se establece que el factorial de 0 es 1. Esto sucede porque el factorial de 0 representa el producto vacío, que por definición da como resultado 1, al ser éste el elemento neutro de

la multiplicación. Así, el caso base es cuando el argumento es 0, pues simplemente se regresa 1 sin necesidad de utilizar recursión.

Con lo anterior podemos dar una definición alternativa de la función factorial haciendo uso de recursión.

$$fact(n) = \begin{cases} 1 & n = 0 \\ n \times fact(n-1) & n > 0 \end{cases}$$

En HASKELL esta definición se ve como en el Código 1.20

```
factorial :: Int -> Int
factorial n | n == 0 = 1
            | n > 0 = n * factorial(n-1)
```

Código 1.20: Definición recursiva de la función `factorial`

La recursión es un mecanismo muy usado, pues no solo permite definir funciones de una manera clara y simple, también facilita probar propiedades sobre éstas haciendo uso de *inducción*.

¡Importante!

Cuando se define una función recursiva se debe asegurar que cada llamada a la función eventualmente llegue al caso base, en caso contrario la función podría no terminar su ejecución.

1.2.14. Coincidencia de patrones

Considérese la función recursiva del Código 1.21 que calcula la longitud de una lista.

```
longitud :: [a] -> Int
longitud xs | null xs   = 0
            | otherwise = 1 + longitud (tail xs)
```

Código 1.21: Función que calcula la longitud de una lista

Observese como en las funciones del Código 1.20 y 1.21 usamos *guardias* para determinar el valor de un argumento y con base en ello, devolver un resultado. Sin embargo, para facilitar esta tarea, en HASKELL se puede hacer uso de expresiones *case*.

Las expresiones *case* tienen la siguiente sintaxis:

```
case expr of
    p1 -> expr1
    p2 -> expr2
    ...
    pn -> exprn
```

en donde tanto *expr* como cada $expr_i$ son expresiones válidas y cada p_i es un *patrón*, que es una forma de determinar la forma de un valor con base en su estructura. Cada p_i `->` $expr_i$ se denomina como *alternativa* [21].

De manera similar a las guardias, la semántica de una expresión *case* evalua en orden cada una de las alternativas, de modo que si el valor de *expr* hace *match*, es decir, coindice con el i-ésimo patrón, entonces el resultado será la i-ésima expresión, en caso contrario se procede a evaluar la siguiente alternativa. Si ningún patrón hace *match* entonces el resultado de toda la expresión será un error.

Es importante mencionar que una expresión *case* debe tener al menos una alternativa y cada una de éstas debe tener el mismo tipo que el resto, así, el tipo de la expresión *case* será dicho tipo.

Existen distintas formas de patrones en HASKELL. Los más simples son:

- **Comodín**: Se denota con un guíon bajo(_). Este patrón se usa cuando la estructura de un valor no es de importancia, por tanto hace *match* con cualquier valor. Es muy útil cuando se usa como última alternativa.
- **Literales**: Es el patrón más simple. Coinciden con un valor específico, como el número 3 o el carácter 'a'.

```
esVocal :: Char -> Bool
esVocal c = case c of
                'a' -> True
                'e' -> True
                'i' -> True
                'o' -> True
                'u' -> True
                _   -> False
```

- **Variables**: El patrón más común, que hace *match* con cualquier valor. A diferencia del comodín, cuando usamos variables podemos hacer uso de ellas en alguna expresión.
- **Constructores**: Hacen *match* con valores que fueron generados por el constructor de un tipo.

```
data RGB = RGB Int Int Int

esRojo :: RGB -> Bool
esRojo (RGB 255 0 0) = True
esRojo _             = False
```

- **Tuplas**: Las tuplas de patrones a la vez son patrones. Por ejemplo, en el Código 1.22 se puede observar la definicion de la función `fst` definida sobre pares.

```
fst :: (a,b) -> a
fst t = case t of
            (x,_) -> x
```

Código 1.22: Definición de la función `fst`

- **Listas**: Las listas son un caso especifico del patrón constructor. Una lista de patrones a la vez es un patrón. Por ejemplo, en el Código 1.23 se puede observar la definición de la función `tiene2Elementos` que determina si una lista tiene exactamente dos elementos.

```haskell
tiene2Elementos :: [Char] -> Bool
tiene2Elementos xs = case xs of
                       [_, _] -> True
                       _      -> False
```

Código 1.23: Función que determina si una lista tiene exactamente dos elementos

En la Sección 1.2.8 se introdujo el operador *cons*, que como se vió, es el constructor usado para construir listas y por ende se puede usar para especificar un patrón. Por ejemplo, en el Código 1.24 se ve la definición de la función `head` que regresa la cabeza de una lista.

```haskell
head :: [a] -> a
head xs = case xs of
            (x:_) -> x
            _     -> error "lista vacía"
```

Código 1.24: Definición de la función `head`

Por otro lado, el patrón que hace *match* con la lista vacía se representa mediante los corchetes vacíos (`[]`). Así, con lo visto hasta ahora, es posible redefinir la función del Código 1.21 usando expresiones *case*.

```haskell
longitud :: [a] -> Int
longitud l = case l of
               []     -> 0
               (_:xs) -> 1 + longitud xs
```

Código 1.25: Definición de la función `longitud` usando expresiones *case*

Como se puede notar, el uso de expresiones *case* es muy conveniente a la hora de definir una función. No obstante, no es la única manera de usar patrones en la definición de una función. Haskell nos permite declarar multiples ecuaciones para la misma función en donde los parámetros de cada una de éstas es un patrón. Por ejemplo, la función del Código 1.25 se puede reescribir como se ve en el Código 1.26

```haskell
longitud :: [a] -> Int
longitud [] = 0
longitud (x:xs) = 1 + longitud xs
```

Código 1.26: Definición de la función `longitud` usando *coincidencia de patrones*

Nota

Al igual que en las expresiones *case*, el orden de definición de las ecuaciones es relevante.

1.2.15. Funciones de order superior

Las funciones en HASKELL son elementos de *primera clase* [7]. Esto significa que las funciones pueden ser tratadas como valores y por ende pueden ser guardadas en estructuras de datos, ser argumentos de otras funciones o ser el resultado de la aplicación de alguna función.

A aquellas funciones que pueden recibir funciones como argumentos o devolver funciones como resultados se les conoce como *funciones de order superior*. Entre las más conocidas estan `map` y `filter`, ambas definidas sobre listas.

La función `map` recibe una función f y una lista xs y devuelve como resultado la lista resultante de aplicarle a cada elemento de xs la función f. Una posible implementación de esta función se ve en el Código 1.27

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

Código 1.27: Definición de la función `map`

```
ghci> map (+10) [1..5]
[11,12,13,14,15]
```

Mientras que la función `filter` recibe un predicado[23] p y una lista xs y como resultado se obtiene la lista de los elementos de xs que cumplan el predicado p.

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
            | p x = x : (filter p xs)
            | otherwise = filter p xs
```

Código 1.28: Definición de la función `filter`

```
ghci> filter even [1..10]
[2,4,6,8,10]
```

1.2.16. Definición de tipos de datos

Además del uso de los tipos primitivos en HASKELL, es posible crear nuestros propios tipos de datos. Para ello, se define la siguiente sintaxis[24]:

$$\texttt{data } \textit{Tipo} = C_1\ t_1^1\ t_1^2\ \ldots \mid C_2\ t_2^1\ t_2^2\ \ldots \mid \ldots \mid C_m\ t_m^1\ t_m^2\ \ldots$$

en donde:

- `Tipo` es el nombre del nuevo tipo de dato que se está definiendo.[25]
- $C_1, C_2, \ldots, C_m$ son los constructores que definen los valores del tipo de dato, con $m \geq 1$.
- $t_i^1, t_i^2, \ldots$ son los tipos de datos que cada constructor puede tomar, los cuales pueded omitirse si el constructor no toma ningún argumento.

[23] Un predicado en HASKELL es una función cuyo valor de retorno es un booleano y es usado para determinar si un valor cumple una cierta condición.

[24] Esta no es la sintaxis completa. Sin embargo para los propósitos del manual se manejará de esta forma.

[25] De la misma manera que con los tipos primitivos, éste debe empezar con una letra mayúscula.

Por ejemplo, considerese la siguiente declaración del tipo `Booleano`[26]

```
data Booleano = Verdadero
              | Falso
              deriving (Show, Eq, Ord)
```

La claúsula `deriving (Show, Eq, Ord)` es opcional y sirve para proporcionar diferentes características[27] a nuestro tipo. Cada una de éstas viene dada por lo que se denomina *Clase de tipos*, que son similares a las *interfaces* en los lenguajes de programación orientados a objetos. Entre las más comunes están:

- `Show`: Permite que el tipo de dato pueda ser representado como cadena.
- `Eq`: Nos permite usar operadores de igualdad (`==`, `/=`) entre los valores de éste tipo.
- `Ord`: Permite dar un órden a los constructores del tipo. Por defecto, el orden está dado por el orden de definición de los constructores. En el caso del tipo `Booleano` se cumple que `Verdadero < Falso`.

Un tipo define un conjunto de posible valores y, en general, con éste vienen asociadas un conjunto de operaciones con el fin de dar un significado. Por ejemplo, habiendo definido el tipo `Booleano` se puede definir la función `negacion` que sea equivalente a la negación lógica.

```
negacion :: Booleano -> Booleano
negacion Falso = Verdadero
negacion Verdadero = Falso
```

Código 1.29: Definición de la función negación sobre el tipo `Boolean`

La sintaxis para definir un nuevo tipo de dato permite que los constructores de éste puedan recibir argumentos. Imagine que se quiere definir un tipo de datos para crear diferentes figuras geométricas como el cuadrado y el triángulo.

[26] Ésta definición es similar a la que del tipo `Bool` que se encuentra en el *Preludio* de Haskell.
[27] Para más información puede revisar la sección *Clases de tipos paso a paso* de la referencia [18].

```haskell
data Figura = Cuadrado Int | Circulo Int
```

Código 1.30: Definición del tipo de dato `Figura`

El Código 1.30 define el tipo `Figura` que puede ser un cuadrado o un círculo. En el caso del cuadrado, se necesita un argumento de tipo `Int` que representa el tamaño de los lados. En el caso del círculo, se necesita un argumento de tipo `Int` que representa el radio.

De esta manera se puede definir un cuadrado usando el respectivo constructor del tipo `Figura`.

```haskell
miCuadrado :: Figura
miCuadrado = Cuadrado 10
```

Por otro lado, se pueden definir funciones que tomen como argumento una figura, como se muestra en el Código 1.31 que calcula el área de una figura.

```haskell
area :: Figura -> Int
area (Cuadrado l) = l * l
area (Circulo r) = 3 * r * r
```

Código 1.31: Definición de la función `area` que calcula el área de una figura

También es posible definir tipos de datos recursivos, de modo que, si se quiere definir a los números naturales usando los axiomas de Peano, se puede hacer de la siguiente forma:

```haskell
data Natural = Cero | Suc Natural
```

Código 1.32: Definición de los números naturales usando los axiomas de Peano

Por tanto, un valor del tipo `Natural` puede ser `Cero` o `Suc n` donde n también es un valor del tipo `Natural`.

Si se quisiera representar al número 2 usando el tipo `Natural`, se haría como sigue:

```
dos :: Natural
dos = Suc (Suc Cero)
```

Así mismo, se puede definir una función que calcule si un número es par o no.

```
esPar :: Natural -> Bool
esPar Cero = True
esPar (Suc Cero) = False
esPar (Suc (Suc n)) = esPar n
```

Código 1.33: Definición de la función `esPar` que calcula si un número es par

Tip

En `GHCi` se puede usar el comando `:i <tipo>` para ver, entre otras cosas, la definición de algún tipo de dato. Por ejemplo `:i Bool`:

```
type Bool :: *
data Bool = False | True -- Defined in `GHC.Types'
...
```

1.2.17. Tipos sinónimo

Si en `GHCi` se ejecuta el comando `:i String`, se obtendría lo siguiente:

```
type String = [Char] -- Defined in `GHC.Base'
```

A diferencia de otros tipos de datos como `Bool`, `Char` o `Int`, el tipo `String` se define por medio de un *tipo sinónimo*.

Los tipos sinónimo no definen tipos nuevos, simplemente renombran a tipos ya existentes.

Por ejemplo, en muchos lenguajes de programación el tipo `Matriz` no es un tipo predefinido.

En su lugar, se usan arreglos de arreglos para representar una matriz. En HASKELL se puede representar una matriz como una lista de listas, por lo que usando un tipo sinónimo se puede definir el tipo `Matriz`

```
type Matriz a = [[a]]
```

En este ejemplo la letra `a` denota una variable de tipo, que significa que `a` puede ser un tipo cualquiera (algo parecido a los *genéricos* en otros lenguajes de programación)[28]. En otras palabras, se pueden crear matrices con elementos de un tipo cualquiera sin necesidad de definir un tipo `Matriz` para cada tipo de dato.

```
identidad :: Matriz Int
identidad = [[1,0][0,1]]
```

Código 1.34: Definición de la matriz identidad de dimensión 2 usando un tipo sinonimo

Los tipos sinónimo se usan para mejorar la legibilidad de los programas.

1.2.18. Módulos

Un programa en HASKELL se conforma, usualmente, por colecciones de módulos. Un *módulo* es una colección de funciones, tipos y clases de tipos que comparten un propósito común. Esto hace que el construir un programa en HASKELL sea en un principio, sencillo, ya que todo se encuentra dividido en módulos, cada uno con su propio propósito.

Técnicamente hablando, un módulo se ve de la siguiente forma

[28]Para más información puede revisar la sección *Variables de tipo* de la referencia [18].

```
module <Nombre> where
-- módulos importados
import <modulo1>
import <modulo2>
...
-- tipos de datos
data MiTipo = ...
-- funciones
foo x = ...
```

Supongase que se quieren implementar el recorrido por profundidad sobre árboles, para el cual se recomienda hacer uso de la estructura de datos conocida como *Pila* [37]. Para ello se puede definir un módulo como el siguiente:

```
module Pila where
    type Pila a = [a]

    push :: a -> Pila a -> Pila a
    push ...
    pop :: Pila a -> a
    pop ...
```

Código 1.35: Definición básica de un módulo

El módulo `Pila` ahora puede ser importado desde cualquier otro módulo:

```
module ArbolBinario where
    import Pila

    data Tree a = Void | Node a (Tree a) (Tree a)

    dfs :: Tree a -> Pila a -> [a]
    dfs t = ...
```

1.3. Desarrollo de la práctica

Resolver los siguiente ejercicios haciendo uso de todo lo presentado a lo largo de la Sección 1.2.

Se prohibe el uso de funciones predefinidas que resuelvan directamente el ejercicio.

1. Definir la función `tercero :: [a] -> a` que regresa el tercer elemento de una lista, que tiene al menos 3 elementos, usando:

   a) Las funciones `head` y `tail`.

   b) *Pattern matching.*

2. Considérese la función `resto :: [a] -> [a]` que se comporta de manera similar a `tail`, que regresa el resto de una lista, con la diferencia de que `resto` devuelve la lista vacía si su argumento es la lista vacía en vez de regresar un error. Usando la función `tail` y la función `null`, ambas vistas en la Sección 1.2.8, definir la función `resto` usando:

   a) *if-then-else.*

   b) Guardias.

   c) *Pattern matching.*

3. Definir la función `mezcla :: [Int] -> [Int] -> [Int]` que recibe como argumentos dos listas ordenadas y regresa la concatenación de éstas preservando el orden. Se espera que la complejidad de la función sea $\mathcal{O}(n + m)$ en dónde n y m son las longitudes correspondientes de cada uno de los argumentos. Es decir, no esta permitido concatenar las listas y luego ordenarlas.

```
ghci> mezcla [1,5,9] [2,8]
[1,2,5,8,9]
```

4. Definir la función `fragmentar :: Int -> [a] -> [[a]]` que toma como argumento un número entero positivo k mayor a 0 y una lista xs y divide a xs en fragmentos de longitud k. El último fragmento puede tener una longitud menor a k si la longitud de

xs no es divisible por k.

```
ghci> fragmentar 3 [1..11]
[[1,2,3],[4,5,6],[7,8,9],[10,11]]
```

5. Definir la función `mapFunc :: [a -> b] -> [a] -> [b]` que toma como argumentos una lista de funciones `fs` y una lista de elementos `xs` y devuelve la lista resultante de aplicar la i-ésima función de `fs` al i-ésimo elemento de `xs`.

```
ghci> mapFunc [(+1),(*2),(^3)] [1,2,3]
[2,4,27]
```

6. Definir la función `esPalindromo :: String -> Bool` que determina si una cadena es un palíndromo. La cadena puede contener espacios en blanco que no deben ser tomados en cuenta. La función no debe distinguir entre mayúsculas y minúsculas, por ejemplo, la palabra *haskell* es igual a *Haskell*.

```
ghci> esPalindromo "Somos o no somos"
True
```

7. Definir la función `prefijoComun :: [String] -> String` que devuelva el prefijo común más largo de un lista de cadenas. En caso de no existir el resultado deberá ser la cadena vacía.

```
ghci> prefijoComun ["funcion", "funcional", "fundamental"]
"fun"
```

8. Definir la función `potencia :: Int -> Int -> Int` que toma como argumentos dos números a y n y computa a^n. La complejidad de la función debe ser $\mathcal{O}(\log\ n)$.
   Sugerencia: Revisar las leyes de los exponentes.

```
ghci> potencia 2 10
1024
```

9. Definir las siguientes funciones sobre el tipo de dato `Natural` presentado en el Código 1.32

   a) `suma :: Natural -> Natural -> Natural` que recibe dos números naturales n y m y devuelve el número natural correspondiente a la suma de n y m.

```
ghci> suma (Suc Cero) (Suc (Suc Cero))
Suc (Suc (Suc Cero))
```

   b) `producto :: Natural -> Natural -> Natural` que recibe dos números naturales n y m y devuelve el número natural correspondiente al producto de n y m.

```
ghci> producto (Suc (Suc Cero)) (Suc (Suc Cero))
Suc (Suc (Suc (Suc Cero)))
```

   c) `toInt :: Natural -> Int` que devuelve el equivalente de un número natural a su versión entera.

```
ghci> toInt (Suc (Suc (Suc Cero)))
3
```

10. Considérese una función f cuyo cuerpo es `[f x | x <- xs, p x]`. Definir una función equivalente a f usando las funciones `map` y `filter`.

Práctica 2

Sintaxis y Semántica de la Lógica Proposicional

2.1. Objetivos

- ▷ Dar una introducción a la Lógica Proposicional desde un punto de vista práctico.
- ▷ Modelar al lenguaje de la Lógica Proposicional haciendo uso de Haskell.

2.2. Introducción

La lógica es una parte fundamental de las Ciencias de la Computación, pues no solo ha ayudado a sentar las bases de éstas, si no que también ha contribuido a su formación. Tanta ha sido la contribución de la lógica a las Ciencias de la Computación que incluso ésta ha sido nombrada como "el cálculo de las Ciencias de la Computación" [19].

Uno de los tantos usos de la lógica en las Ciencias de la Computación es desarrollar lenguajes que ayuden a modelar problemas, de manera que se pueda razonar formalmente sobre ellos, esto es, construir argumentos y conclusiones que puedan ser defendidos de manera rigurosa.

El sistema lógico más simple es el de la lógica proposicional, que está basado en la especificación formal y uso de proposiciones, que son sentencias, de las cuales se puede argumentar si son verdadera o falsas. A modo de ejemplo, considérese las siguientes proposiciones:

- Haskell es un lenguaje de programación funcional.

- Cualquier número natural mayor a 2 puede ser expresado como la suma de dos números primos.
- La complejidad en tiempo del algoritmo *Quicksort* es $\mathcal{O}(n^2)$.

Para representar estas proposiciones, el sistema de la Lógica Proposicional describe un lenguaje, el cual no trabaja directamente con las sentencias, sino que éstas se codifican en una cadena de símbolos, obteniendo una representación comprimida pero completa de las proposiciones. Permitiendo así, concentrarse en la estructura de éstas. Nótese cómo el trabajar con símbolos abre la posibilidad de manipular éstos de manera automática haciendo uso de una computadora.

2.2.1. Sintaxis

Como se mencionó, el sistema de la Lógica Proposicional viene descrito por medio de un lenguaje, el cual consta del siguiente alfabeto:

- Variables proposicionales: $p_1, p_2, \ldots, p_n, \ldots$
- Conectivos lógicos: $\neg, \vee, \wedge, \rightarrow, \leftrightarrow$
- Constantes lógicas[1]: $\top, \bot$
- Parentesis: (,)

Al conjunto de variables proposicionales y constantes lógicas se le conoce como fórmulas atómicas, y se denota como ATOM.

Una vez definido el alfabeto se procede a definir las reglas que permiten determinar si una cadena pertenece o no al lenguaje, es decir, la sintaxis.

Definición 1 (Lenguaje PROP) *Las expresiones bien formadas, usualmente conocidas como fórmulas, que pertenecen al lenguaje de la Lógica Proposicional, denotado como* PROP*, se definen recursivamente como sigue [9]:*

- *Si $p \in ATOM$ entonces p es una proposición.*

[1]También conocidas como *Verdadero* y *Falso* respectivamente.

- *Si p es una fórmula proposicional entonces $(\neg p)$ es una fórmula proposicional.*
- *Si p y q son fórmulas proposicionales entonces $(p \vee q)$, $(p \wedge q)$, $(p \to q)$, $(p \leftrightarrow q)$ son fórmulas proposicionales.*
- *Son todas.*

Nótese como la definición anterior al ser recursiva permite describir el lenguaje por medio de una gramática:

```
Atom := Variable | ⊤ | ⊥
Prop := Atom
        | ¬ Prop
        | Prop ∧ Prop
        | Prop ∨ Prop
        | Prop → Prop
        | Prop ↔ Prop
```

Cuya traducción a HASKELL se ve como en el Código 2.1.

```
data Prop =   Var String | Cons Bool | Not Prop
            | And Prop Prop | Or  Prop Prop
            | Impl Prop Prop | Equiv Prop Prop
            deriving (Eq, Show)
```

Código 2.1: Definición del tipo de dato `Prop`.

Nota

Puesto que todo elemento de ATOM es elemento de PROP, se incluyen los constructores de ATOM en `Prop` para simplificar la definición del tipo de datos.

Habiendo definido el tipo de dato `Prop`, ahora es posible crear expresiones de la lógica proposicional, por ejemplo, la proposición $(p \wedge q) \vee r \to p$ se ve como en el Código 2.2.

```
let p = Var "p"; q = Var "q"; r = Var "r" in Impl (Or (And p q) r) p
```

Código 2.2: Ejemplo que demuestra el uso del tipo de dato `Prop`

2.2.2. Precedencia de operadores

Como se puede observar en el Código 2.2, la creación y manipulación de estas expresiones puede ser un poco tediosa debido al uso de los parentesis. Para evitar el uso de éstos, se define la siguiente precedencia de operadores, de mayor a menor:

- $\neg$
- $\vee, \wedge$
- $\rightarrow$
- $\leftrightarrow$

Por otro lado, los conectivos $\vee, \wedge, \leftrightarrow$ son asociativos a la izquierda, y aunque el conectivo $\rightarrow$ no es asociativo[2], por convención se define como asociativo a la derecha [9].

Tip

En vez de usar el constructor `And` del tipo `Prop` se puede definir un operador que sea equivalente a éste.

```
(/\) :: Prop -> Prop -> Prop
(/\) = And

ghci> let p = Var "p"; q = Var "q" in p /\ q == p `And` q
True
```

Usando la precedencia de operadores antes definida, la expresión $p \rightarrow q \vee r$ es equivalente a $p \rightarrow (q \vee r)$. Sin embargo se observa que con la definición dada en Haskell no ocurre de esa manera.

[2]La no asociatividad de la implicación se debe a que la forma en como se agrupan los elementos de la expresión afecta el resultado final. Por ejemplo, la expresión $p \rightarrow q \rightarrow r$ puede interpretarse como $(p \rightarrow q) \rightarrow r$ o como $p \rightarrow (q \rightarrow r)$, lo que lleva a diferentes evaluaciones. Si p, q y r son falsas entonces la primera interpretación es falsa, mientras que la segunda es verdadera.

```
ghci> let p = Var "p"; q = Var "q"; r = Var "r"
      in p `Impl` q `Or` r
Or (Impl (Var "p") (Var "q")) (Var "r") -- (p -> q) \/ r
```

Con lo visto en la Sección 1.2.5, se pueden definir reglas de asociatividad y precedencia para los constructores del tipo `Prop`, las cuales se definen[3]como en el Código 2.3.

```
infixl 5  `And`,`Or`
infixl 4  `Equiv`
infixr 3  `Impl`
```

Código 2.3: Definición de la precedencia de operadores.

Con lo anterior, ahora es posible definir fórmulas sin usar parentesis.

```
ghci> let p = Var "p"; q = Var "q"; r = Var "r"
      in p `Impl` q `Or` r
Impl (Var "p") (Or (Var "q") (Var "r")) -- p -> (q \/ r)
```

2.2.3. Sustitución textual

La sustitución textual es una operación muy importante en la lógica, ya que permite manipular y transformar fórmulas lógicas sin cambiar el significado de ellas. Ésta es una transformación sintáctica que consiste en reemplazar una expresión por otra.

La aplicación de una sustitución textual a una expresión se denota como $\varphi[x := \psi]$, donde x es una variable y tanto φ como ψ son fórmulas, y se define como la expresión que resulta de reemplazar todas las ocurrencias de x en φ por ψ. La definición recursiva de la sustitución textual es como sigue:

[3]El uso de los números 3,4,5 para definir la precedencia de los operadores fue elegido de manera arbitraria. En realidad se pudieron elegir 3 números en orden ascendente en el rango de 1 a 9.

$$
\begin{aligned}
p[p := \psi] &= \psi \\
q[p := \psi] &= q, \text{ si } p \neq q \\
\top[p := \psi] &= \top \\
\bot[p := \psi] &= \bot \\
(\neg\varphi)[p := \psi] &= \neg(\varphi[p := \psi]) \\
(\varphi \wedge \chi)[p := \psi] &= (\varphi[p := \psi] \wedge \chi[p := \psi]) \\
(\varphi \vee \chi)[p := \psi] &= (\varphi[p := \psi] \vee \chi[p := \psi]) \\
(\varphi \rightarrow \chi)[p := \psi] &= (\varphi[p := \psi] \rightarrow \chi[p := \psi]) \\
(\varphi \leftrightarrow \chi)[p := \psi] &= (\varphi[p := \psi] \leftrightarrow \chi[p := \psi])
\end{aligned}
$$

En HASKELL se puede definir una sustitución por medio de un sinónimo de tipo como se muestra en el Código 2.4.

```
type Sustitucion = (String, Prop)
```

Código 2.4: Definición del tipo de dato `Sustitucion`.

De esta manera, el tipo de la función de sustitución es como en el Código 2.5.

```
sustituye :: Prop -> Sustitucion -> Prop
```

Código 2.5: Firma de la función de sustitución.

2.2.4. Semántica

Hasta ahora solo se ha definido la *forma* del lenguaje que se esta estudiando, es decir, las reglas sintácticas que permiten decidir si una fórmula proposicional esta bien formada. Sin embargo, es por medio de una semántica que se le da un significado a estas fórmulas.

2.2.5. Interpretaciones

La semántica de las fórmulas proposicionales se define usualmente respecto a un estado de las variables, que no es más que una asignación de valores de verdad (*verdadero, falso*) a cada proposición de éstas.

Concretamente, en HASKELL podemos representar este estado como una lista de tuplas, como

se puede observar en el Código 2.6, en donde la primera proyección de cada elemento de la lista corresponde al identificador de cada proposición, mientras que la segunda proyección se refiere al valor asignado.

```
type Interpretacion = [(String, Bool)]
```

Código 2.6: Definición de una *asignación* de valores de verdad.

Una asignación de valores de verdad solo nos permite conocer el significado de una proposición. No obstante, se pretende trabajar con fórmulas, que son estructuras más complejas construidas a base de fórmulas más pequeñas y conectadas entre sí por medio de conectivos lógicos. Por tanto es importante definir una función que determine el valor de verdad de una fórmula respecto a un estado de las variables. A esta función se le conoce como *interpretación* y su definición es la siguiente [9]:

- $\mathcal{I}^\star(p) = \mathcal{I}(p)$
- $\mathcal{I}^\star(\top) = 1$
- $\mathcal{I}^\star(\bot) = 0$
- $\mathcal{I}^\star(\neg\varphi) = 1$ *syss* $\mathcal{I}^\star(\varphi) = 0$.
- $\mathcal{I}^\star(\varphi \wedge \psi) = 1$ *syss* $\mathcal{I}^\star(\varphi) = \mathcal{I}^\star(\psi) = 1$.
- $\mathcal{I}^\star(\varphi \vee \psi) = 0$ *syss* $\mathcal{I}^\star(\varphi) = \mathcal{I}^\star(\psi) = 0$.
- $\mathcal{I}^\star(\varphi \rightarrow \psi) = 0$ *syss* $\mathcal{I}^\star(\varphi) = 1$ ***e*** $\mathcal{I}^\star(\psi) = 0$.
- $\mathcal{I}^\star(\varphi \leftrightarrow \psi) = 1$ *syss* $\mathcal{I}^\star(\varphi) = \mathcal{I}^\star(\psi)$.

En donde $\mathcal{I}$(p) denota el valor de verdad asignado a p bajo el estado $\mathcal{I}$.

2.2.6. Modelos y Satisfacibilidad

Si bajo un estado de las variables r, la interpretación de una fórmula ψ es *verdadera*, entonces se dice que r es *modelo* de ψ.

Del concepto de *modelo*, surgen algunos conceptos semánticos de importancia: [9]

- Si dada una fórmula ψ sucede que para todo posible estado de las variables, es modelo

de ψ, entonces se dice que la fórmula es una *tautología*.

- Si dada una fórmula ψ sucede que para todo posible estado de las variables, no es modelo de ψ, entonces se dice que la fórmula es una *contradicción*.
- Si para algún estado de las variables r, la interpretación de una fórmula ψ es *verdadero*, entonces decimos que ψ es satisfacible en r.
- Si para algún estado de las variables r, la interpretación de una fórmula ψ es *falsa*, entonces decimos que ψ es insatisfacible en r.

De manera análoga, si tenemos un conjunto de fórmulas Γ decimos que: [9]

- Γ es satisfacible si existe alguna estado de las variables r, en donde para toda $\psi \in \Gamma$, la interpretación de ψ sea *verdadera*.
- Γ es insatisfacible si para todo estado de las variables r y para toda $\psi \in \Gamma$, la interpretación de ψ es *falsa*.

De las definiciones anteriores se deriva un concepto semántico muy importante dentro de la Lógica Proposicional el cual lleva por nombre *consecuencia lógica*, y sirve para determinar la validez de un argumento.

Definición 2 (Consecuencia Lógica) *Sean Γ un conjunto de fórmulas y ψ una fórmula. Se dice que ψ es consecuencia lógica de Γ si para toda interpretación que satisface a Γ sucede que satisface también a ψ. [9]*

2.3. Desarrollo de la práctica

Con base en la definición del tipo de dato `Prop` que se muestra en el Código 2.1 definir las siguientes funciones:

> ***Nota***
>
> Para simplificar los ejemplos mostrados en cada ejercicio, se decidió omitir las declaraciones `let` para las variables proposicionales. Así, en vez de escribir `let p = Var "p"; q = Var "q"in (p `And` q)`, se escribe solamente `(p `And` q)`, asumiendo que las declaraciones de `p` y `q` se hicieron previamente.

1. Definir una instancia de la clase `Show` para el tipo de dato `Prop`. El resultado debe ser una cadena de caracteres que representa la expresión proposicional en notación infija.
   Nota: En algunos casos es necesario agregar paréntesis para evitar ambigüedades.

```
instance Show Prop where
    show (Var p) = p
    show ...

ghci> (p `And` q) `Or` r
"(p /\ q) \/ r"
```

2. Definir la función `vars :: Prop -> [String]` que devuelve el conjunto (representado como una lista sin repeticiones) de las variables proposicionales de una fórmula.
   Nota: El orden del resultado no importa.

```
ghci> vars (p `And` q `Impl` r `Or` q)
["p", "q", "r"]
```

3. Usando la definición del tipo `Sustitucion` mostrada en el Código 2.4 definir la función `sust :: Prop -> Sustitucion -> Prop` que realice la sustitución textual correspondiente.

```
ghci> sust (p `Impl` q `And` r) ("q", s `And` t)
"p -> s /\ t /\ r"
```

4. Definir la función `interpreta :: Prop -> Estado -> Bool` que dada una fórmula y un estado de las variables regrese la interpretación de ésta.

```
ghci> interpreta (r `Impl` p `And` q) [(p,True),(q,True),(r,False)]
True
```

5. Definir la función `modelos :: Prop -> [Estado]` que dada una fórmula devuelva una lista de los estados que satisfacen a ésta.

   Nota: El orden del resultado no importa.

```
ghci> modelos (p `Or` Not q)
[[("p",True),("q",True)],
[("p",True),("q",False)],
[("p",False),("q",False)]]
```

6. Se dice que dos fórmulas `p` y `q` son lógicamente equivalentes si para todo estado de las variables $\mathcal{I}$ sucede que $\mathcal{I}(p) = \mathcal{I}(q)$. Definir la función `equiv :: Prop -> Prop -> Bool` que determine si dos fórmulas son lógicamente equivalentes.

```
ghci> equiv (Not (p `And` q)) ((Not q) `Or` (Not p))
True
```

7. Definir la función `tautologia :: Prop -> Bool` que determine si una fórmula es tautología.

```
ghci> tautologia (Not (p `Or` q) `Equiv` ((Not p) `And` (Not q)))
True
```

8. Definir la función `contradiccion :: Prop -> Bool` que determine si una fórmula es contradicción.

```
ghci> contradiccion ((p `Impl` q) `And` p `And` (Not q))
True
```

9. Definir la función `satisfacible :: Prop -> Bool` que determine si una fórmula es satisfacible.

```
ghci> satisfacible (Not (p `Or` q) `Equiv` ((Not p) `And` q))
True
```

10. Definir la función `insatisfacible :: Prop -> Bool` que determine si una fórmula es insatisfacible.

```
ghci> insatisfacible ((p `Impl` q) `And` p `And` (Not q))
True
```

11. Definir la función `satisfacibleConj :: [Prop] -> Estado -> Bool` que determine si un conjunto de fórmulas es satisfacible bajo un determinado estado de las variables.

```
ghci> satisfacibleConj [p `Impl` r, q `Impl` s, (Not r) `Or` (Not
   s)] [("p", False), ("q", False), ("r", False), ("s", False)]
True
```

12. Definir la función `satConj :: [Prop] -> Bool` que determine si un conjunto de fórmulas es satisfacible.

```
ghci> satConj [p `Impl` r, q `Impl` s, (Not r) `Or` (Not s)]
True
```

13. Definir la función `insatisfacibleConj :: [Prop] -> Estado -> Bool` que determine si un

conjunto de fórmulas es insatisfacible bajo un determinado estado de las variables.

```
ghci> insatisfacibleConj [p `Impl` q, p, Not q] [(p, True), (q,
   False)]
True
```

14. Definir la función `insatConj :: [Prop] -> Bool` que determine si un conjunto de fórmulas es insatisfacible.

```
ghci> insatConj [p `Impl` q, p, Not q]
True
```

15. Definir la función `consecuencia :: [Prop] -> Prop -> Bool` que dado un conjunto de fórmulas Γ y una fórmula ϕ determine si ϕ es consecuencia lógica de Γ.
    Sugerencia: *Revisar el principio de refutación.*

```
ghci> consecuencia [p `Impl` r, q `Impl` s, (Not r) `Or` (Not s)]
   ((Not p) `Or` (Not q))
True
```

Práctica 3

Resolución Binaria

3.1. Objetivos

- ▷ Revisar el concepto de *Forma Normal* en la Lógica Proposicional desde un punto de vista práctico.
- ▷ Hacer uso de la resolución binaria para decidir la satisfacibilidad de una fórmula proposicional.
- ▷ Implementar un algoritmo de saturación.

3.2. Introducción

Además de las interpretaciones, existen otros métodos que son más eficientes para decidir la satisfacibilidad de una fórmula proposicional. En el Capítulo 2 se vio que para determinar si una fórmula es satisfacible, se debe buscar una interpretación que sea modelo de ésta. Sin embargo, en la práctica esto es muy ineficiente, ya que el número de interpretaciones crece exponencialmente respecto al número de variables proposicionales de la fórmula.

Existen métodos más eficientes para decidir la satisfacibilidad de una fórmula proposicional, los cuales se basan en una regla de inferencia muy importante en la Lógica Proposicional conocida como Resolución Binaria[1]. Es tal la importancia de esta regla de inferencia que se ha demostrado que usando resolución binaria es posible construir un demostrador automático

de teoremas que sea correcto y completo para la Lógica Proposicional [10].

3.2.1. Formas Normales

La resolución binaria no trata directamente con formulas proposicionales. En cambio, trabaja la forma clausular de éstas. En Lógica Proposicional, una claúsula se define como una disyunción de literales, en dónde una literal es o una variable o su negación.

Para hacer que la implementación sea más legible se definen los siguientes sinónimos de tipo:

```
type Literal = Prop
...
ghci> let p = Var "p" :: Literal
ghci> let q = Var "q" :: Literal
```

Código 3.1: Definición del tipo `Literal`

```
type Clausula = [Literal]
...
ghci> let c = [p, q] :: Clausula
ghci> let d = [r, Neg q] :: Clausula
```

Código 3.2: Definición del tipo `Clausula`

Tenga en cuenta que la lista vacía también es una cláusula, que es equivalente a una disyunción vacía y, por tanto, es insatisfacible. Como se verá, se trata de un caso especial particularmente importante.

Usando equivalencias lógicas es posible obtener la forma clausular de cualquier fórmula proposicional.

Forma Normal Negativa

La primera transformación que se le aplica a la fórmula es la *forma normal negativa* (FNN).

[1]También conocida como Resolución Proposicional o Princípio de Resolución [10].

Definición 3 (Forma Normal Negativa) *Una fórmula proposicional φ esta en forma normal negativa si [9]:*

- *φ no contiene equivalencias ni implicaciones.*
- *Las negaciones solo afectan a variables proposicionales.*

Cualquier fórmula proposicional puede ser convertida a *forma normal negativa* en tiempo lineal usando las siguientes equivalencias lógicas.

- Eliminación de la doble negación: $\neg\neg\varphi \equiv \varphi$
- Leyes de Morgan:
  - $\neg(\varphi \wedge \rho) \equiv \neg\varphi \vee \neg\rho$
  - $\neg(\varphi \vee \rho) \equiv \neg\varphi \wedge \neg\rho$

Forma Normal Conjuntiva[2]

Definición 4 (Literal) *Una literal es una fórmula atómica (una variable proposicional o una constante lógica) o la negación de una fórmula atómica [9].*

Definición 5 (Claúsula) *Una cláusula es una literal o una disyunción de literales [9].*

Definición 6 (Forma Normal Conjuntiva) *Una fórmula proposicional φ esta en formal normal conjuntiva si es una conjunción de disyunciones de literales, es decir, si es una conjunción de cláusulas [9].*

Cualquier fórmula proposicional puede ser convertida a *forma normal conjuntiva* en tiempo exponencial[3] usando las siguientes equivalencias lógicas.

[2]También xiste la forma normal disyuntiva (FND), la cual es la forma dual de la FNC. La FND es una disyunción de conjunciones de literales. No obstante, no es de interés para este trabajo.

[3]Se puede hacer la transformación en tiempo lineal usando transformaciones de Tseitin [36], obteniendo una fórmula equisatisfacible, pero no equivalente. Ésta introduce nuevas variables y cláusulas para representar la estructura de la fórmula original y luego aplica una serie de equivalencias lógicas para simplificar la fórmula resultante.

- Eliminación de la doble negación: $\neg\neg\varphi \equiv \varphi$
- Leyes de Morgan:
  - $\neg(\varphi \wedge \rho) \equiv \neg\varphi \vee \neg\rho$
  - $\neg(\varphi \vee \rho) \equiv \neg\varphi \wedge \neg\rho$
- Leyes distributivas:
  - $\varphi \wedge (\rho \vee \psi) \equiv (\varphi \wedge \rho) \vee (\varphi \wedge \psi)$
  - $\varphi \vee (\rho \wedge \psi) \equiv (\varphi \vee \rho) \wedge (\varphi \vee \psi)$

Existe una notación conocida como *forma clausular* que permite representar una fórmula en forma normal conjuntiva de manera compacta. En esta notación, una fórmula en forma normal conjuntiva se representa como un conjunto de conjuntos de literales. Cada conjunto de literales representa una cláusula. Por ejemplo, la fórmula $(p \vee q) \wedge (\neg p \vee r)$ se representa como:

$$\{\{p, q\}, \{\neg p, r\}\}$$

3.2.2. Resolución Binaria

La idea de la resolución binaria es simple. Supóngase que se tiene el siguiente conjunto de claúsulas:

$$\{\{p, q\}, \{\neg q, r\}\}$$

Para que el conjunto sea satisfacible ambas cláusulas deben ser satisfacibles. En el caso de la primer claúsula, se debe dar el caso de que p sea verdadera o q sea verdadera. En el caso de la segunda cláusula, se debe dar el caso de que q sea falsa o r sea verdadera. Si ocurre que q es verdadera, entonces p debe ser verdadera. Si q es falsa, entonces r debe ser verdadera. En cualquier caso, p debe ser verdadera o r debe ser verdadera. Por lo tanto, el problema se reduce a determinar si la cláusula $\{p, r\}$ es satisfacible.

Lo anterior es la base de la regla de inferencia que se muestra a continuación. Dada una cláusula que contiene la literal ℓ y otra cláusula que contiene la literal contraria $\neg\ell$, se puede inferir una nueva claúsula que consta de las literales de dichas claúsulas sin el par complementario, a la cual se le denomina *resolvente*.

$$\frac{C_1 \vee l \quad C_2 \vee \neg l}{C_1 \vee C_2}$$

Tenga en cuenta que, dado que las cláusulas son conjuntos de literales, no puede haber dos ocurrencias de ninguna literal en una cláusula. Por lo tanto, el resolvente debe seguir cumpliendo la definición de conjunto.

$$\frac{\{\neg p, q\} \quad \{p, q\}}{\{q\}}$$

A las claúsulas que constan de una sola literal se les llama *unitarias*. De modo que si al aplicar resolución binaria una de las dos claúsulas involucradas es unitaria entonces el resolvente contendra las literales restantes de la otra claúsula.

$$\frac{\{p, q, r\} \quad \{\neg p\}}{\{q, r\}}$$

El resolvente de dos claúsulas unitarias es conocido como claúsula vacía, pues no contiene ninguna literal. Obtener ésta como resultado indica que el conjunto de claúsulas no es satisfacible.

$$\frac{\{p\} \quad \{\neg p\}}{\square}$$

Es importante destacar que la regla solo permite que se haga una resolución a la vez, en otras palabras, solo es posible aplicar la regla sobre un par de literales a la vez.

$$\frac{\{p, q\} \quad \{\neg p, \neg q\}}{\square} \times \qquad \frac{\{p, q\} \quad \{\neg p, \neg q\}}{\{q\}} \checkmark$$

3.2.3. Algoritmo de Saturación

Razonar con el Principio de Resolución es análogo a razonar con otras reglas de inferencia. Se comienza con algunas premisas; se aplica el principio de resolución a esas premisas; aplicamos la regla a los resultados de esas aplicaciones; y así sucesivamente hasta obtener una conclusión [10]. Este procedimiento se puede abstraer en un *algoritmo de Saturación.*

Definición 7 *[9] Sea* $\mathbb{S}$ *un conjunto de cláusulas.*
Sea $\mathbb{R}(\mathbb{S})$ *el conjunto resultado de la unión de* $\mathbb{S}$ *con el conjunto de todos los posibles resolventes de las cláusulas de* $\mathbb{S}$*.*

$$\mathbb{R}(\mathbb{S}) = \mathbb{S} \bigcup \{res(c_1, c_2) : c_1, c_2 \in S\}$$

en donde res es la función que calcula el resolvente de dos cláusulas.

Se define la n-ésima resolución de $\mathbb{S}$ *como*

$$Res_0(\mathbb{S}) = \mathbb{S}$$
$$Res_{n+1}(\mathbb{S}) = \mathbb{R}(Res_n(\mathbb{S}))$$

De manera que si existe una $n \in \mathbb{N}$ tal que $\square \in Res_n(\mathbb{S})$ se concluye que $\mathbb{S}$ es insatisfacible.

Nótese como lo anterior nos da un método para verificar si un conjunto de cláusulas $\mathbb{S}$ es insatisfacible, pues, al generar la n-ésima resolución de $\mathbb{S}$ se tiene uno de los siguientes casos:

1. $\square \in Res_n(\mathbb{S})$, entonces como se encontró la claúsula vacía se concluye que $\mathbb{S}$ es insatisfacible.
2. $Res_{n+1}(\mathbb{S}) = Res_n(\mathbb{S})$, si sucede que ya no es posible generar más resolventes y no se encontró la claúsula vacía en alguna iteración, entonces se concluye que $\mathbb{S}$ es satisfacible.
3. La ejecución del algoritmo termina hasta que se agoten los recursos computacionales, concluyendo que se desconoce si $\mathbb{S}$ es satisfacible o no.

En Haskell este algoritmo puede ser implementado mediante la siguiente función:

```
saturacion :: [Clausula] -> Bool
```

la cual recibe una función en forma clausular, es decir, una lista de claúsulas, y determina si el conjunto de claúsulas es satisfacible o no.

Es importante mencionar que para esta implementación la claúsula vacía se representa como la lista vacía.

3.3. Desarrollo de la práctica

1. Definir la función `fnn :: Prop -> Prop` que dada una fórmula, devuelva la forma normal negativa de la misma.

```
ghci> fnn  (p `Equiv` q)
((Not p `Or` q) `And` (Not q `Or` p))
```

2. Definir la función `fnc :: Prop -> Prop` que dada un fórmula, devuelva la forma normal conjuntiva de la misma.

```
ghci> fnc (p `And` (q `Impl` r))
(p `And` (Not q `Or` r))
ghci> fnc  (Not (p `And` (q `Impl` r)))
((Not p `Or` q) `And` (Not p `Or` Not r))
```

3. Definir la función `clausulas :: Prop -> [Clausula]` que dada un fórmula en forma normal conjuntiva, devuelve una lista con las claúsulas que la forman.

```
ghci> clausulas  (((Not p) `Or` q) `And` ((Not p) `Or` (Not r)))
[[q, Not p],[Not p, Not r]]
```

4. Definir la función `hayResolvente :: Clausula -> Clausula -> Bool` que determina si es posible obtener un resolvente a partir de dos claúsulas.

```
ghci> hayResolvente [p, r, s] [Not p, Not q, s]
True
ghci> hayResolvente [p, r, s] [p, q, r, Not t]
False
```

5. Definir la función `resolucion :: Clausula -> Clausula -> Clausula` que dadas dos claúsulas, devuelve el resolvente obtenido después de aplicar la regla de *resolución binaria*.

Se puede asumir que se puede obtener un resolvente a partir de los argumentos.

```
ghci> resolucion  [Not p, r, q, Not s] [p, r, q, Not s]
[r, q, Not s]
```

6. Definir la función `saturacion :: [Clausula] -> Bool` que dado un conjunto de claúsulas, determina si éste es satisfacible o no usando el algoritmo de Saturación.

```
ghci> saturacion [[r, p, q], [r], [Not q]]
True
ghci> saturacion [[Not p],[Not q, p],[Not r, p],[q, r]]
False
```

7. Modifica la función anterior para que reciba un argumento adicional que sea un número entero positivo n que indica el número máximo de pasos que se pueden realizar en el algoritmo de Saturación. Si es imposible determinar la satisfacibilidad antes de llegar al límite, entonces la respuesta deberá ser `False`.

Práctica 4

Lógica de Primer Orden

4.1. Objetivos

- Dar una introducción a la Lógica de Primer Orden desde un punto de vista práctico.
- Modelar el lenguaje de la Lógica de Primer Orden usando Haskell.
- Implementar algunas funciones sintácticas de la Lógica de Primer Orden.
- Implementar un algoritmo de unificación.

4.2. Introducción

Aunque la Lógica Proposicional es un buen punto de partida para describir los principios generales del razonamiento lógico, ésta cuenta con algunas limitaciones. Por ejemplo, tómese en cuenta el siguiente acertijo lógico [35]:

> *Alice, el esposo de Alice, su hijo, su hija y el hermano de Alice estuvieron involucrados en un asesinato. Uno de los cinco mató a uno de los otros cuatro. Los siguientes hechos se refieren a las cinco personas mencionadas:*
>
> - *Un hombre y una mujer estaban juntos en un bar en el momento del asesinato.*
> - *La víctima y el asesino estaban juntos en una playa en el momento del asesinato.*
> - *Uno de los dos hijos de Alice estaba solo en el momento del asesinato.*

- *Alice y su esposo no estaban juntos en el momento del asesinato.*
- *El gemelo de la víctima no era el asesino.*
- *El asesino era más joven que la víctima.*

¿Cuál de los cinco fue la víctima?

Para expresar principios como estos, se necesita una forma de hablar sobre objetos e individuos, así como sobre sus propiedades y las relaciones entre ellos. La Lógica Proposicional no brinda los medios para expresar un principio general que establezca que si Alice está con su hijo en la playa, entonces su hijo está con Alice; el hecho de que ningún niño es mayor que su padre; o el hecho de que si alguien está solo, entonces no está con otra persona. Esto es exactamente lo que proporciona el sistema lógico conocido como Lógica de Primer Orden[1].

4.2.1. Sintaxis

En la Lógica Proposicional, las fórmulas atómicas no tienen estructura interna: son variables proposicionales a las que se les asigna el valor de *verdadero* o *falso*. En la Lógica de Primer Orden, las fórmulas atómicas son *predicados* que afirman una relación entre ciertos elementos. Tanto los elementos como las relaciones se construyen sobre un *universo de discurso*, que es un conjunto de objetos que se consideran relevantes para el problema en cuestión.

Otro concepto nuevo de importancia en la Lógica de Primer Orden es la *cuantificación*, que es la capacidad de afirmar que cierta propiedad se cumple para todos los elementos o que se cumple para algunos.

A diferencia de la Lógica Proposicional, la Lógica de Primer Orden no tiene un lenguaje fijo, pues este depende de una *signatura*, la cual se conforma de los siguientes conjuntos, que son ajenos entre sí [9]:

- símbolos de constantes: $a,b,c,d,\ldots$
- símbolos de funciones: $f,g,h,\ldots$

[1]También conocida como Lógica de Predicados o Cálculo de Predicados.

- símbolos de predicados: $P, Q, R, \ldots$

Independientemente de la signatura del lenguaje, se cuenta con el siguiente alfabeto:

- Un conjunto infinito de variables $\{x_0, x_1, x_2, \ldots\}$
- Constantes lógicas: $\top, \bot$
- Conectivos lógicos: $\neg, \wedge, \vee, \rightarrow, \leftarrow, \leftrightarrow$
- Cuantificadores: $\forall, \exists$
- Parentesis: (,)

Dada una signatura σ, se define el lenguaje de *términos* (**TERM**), de manera inductiva como sigue [9]:

- Una variable es un término.
- Una constante es un término.
- Si $t_1, \ldots, t_k$ son términos, y f es una función de aridad k entonces $f(t_1, \ldots, t_k)$ es un término.

Es importante mencionar que estos términos se definen respecto a un universo de discurso, el cual es un conjunto no vacío de elementos.

Para modelar este lenguaje en HASKELL, se define el tipo de dato `Term` como se muestra en el Código 4.1

```
data Term = Var String | Fun String [Term] deriving (Eq, Show)
```

Código 4.1: Definición del tipo de dato `Term`.

Nótese que no existe un constructor para las constantes. Esto se debe a que las constantes pueden modelarse como una función de aridad 0.

Habiendo definido el lenguaje de términos, se define el lenguaje de fórmulas de manera inductiva como sigue [9]:

- Las constantes lógicas ($\top, \bot$) son fórmulas.
- Si $t_1, \ldots, t_k$ son términos y P es un símbolo de predicado de aridad k entonces $P(t_1, \ldots, t_k)$

es una fórmula.

- Si φ es una fórmula, entonces $\neg\varphi$ es una fórmula.
- Si φ y ψ son fórmulas entonces $(\varphi \vee \psi)$, $(\varphi \wedge \psi)$, $(\varphi \rightarrow \psi)$, $(\varphi \leftrightarrow \psi)$ también son fórmulas.
- Si φ es una fórmula y x una variable entonces $\exists x\varphi$ y $\forall x\varphi$ también son fórmulas.

Y mediante el tipo de dato `Formula`, se define el lenguaje de fórmulas como se observa en el Código 4.2

```
type Simbolo = String
data Formula = Top | Bottom
             | Predicado Simbolo [Term]
             | Not Formula
             | And Formula  Formula  | Or Formula  Formula
             | Impl Formula  Formula  | Iff Formula  Formula
             | ForAll Simbolo Formula  | Exists Simbolo Formula
             deriving (Eq, Show)
```

Código 4.2: Definición del tipo de dato `Formula`.

Los cuantificadores tienen menor precedencia que cualquier conectivo lógico [9]. Por ello es importante definir dicha precedencia al momento de definir los tipos de datos.

```
infixr 4  `ForAll`, `Exists`
infixl 3  `And`,`Or`
infixl 2  `Iff`
infixr 1  `Impl`
```

A modo de ejemplo, y para comprender el uso del tipo de dato `Formula`, tómese en cuenta el universo de los números naturales y la siguiente declaración:

> *Cada número natural mayor a 1 tiene al menos un divisor que es un número primo.*

Para formalizar lo anterior usando Lógica de Predicados primero se definen los predicados,

funciones y constantes a usar.

- G(x,y): x es mayor que y.
- Primo(x): x es un número primo.
- Div(x,y): x es divisible por y.

Por tanto, la formalización de la declaración anterior es:

$$\forall n(G(n,1) \rightarrow \exists p(Primo(p) \wedge Div(n,p)))$$

Que al traducirlo a HASKELL usando el tipo de dato `Formula` se tiene:

```
p = ForAll "n" (Predicado "G" [Var "n", Fun "1" []] `Imp`
                Exists "p" ((Predicado "Primo" [Var "p"]) `And`
                   (Predicado "Div" [Var "n",Var "p"])))
```

4.2.2. Sustitución

A diferencia de la Lógica Proposicional, la sustitución en la Lógica de Primer Orden no es una operación textual, pues dada la existencia de términos y variables ligadas[2], al aplicar una sustitución a una fórmula, el significado de ésta puede cambiar.

La aplicación de una sustitución textual a una expresión, al igual que en la lógica proposicional, se denota como $\varphi[x := \psi]$, donde x es una variable y tanto φ como ψ son fórmulas, y se define como la expresión que resulta de reemplazar todas las ocurrencias de x en φ por ψ. Primero véase como funciona la sustitución sobre terminos[3][9].

[2]Dada una fórmula de la forma $\exists x\varphi$ o $\forall x\varphi$, si x figura en φ se dice que es una variable ligada.

[3]La sustitución sobre términos si es textual, pues no existen variables ligadas.

$$
\begin{aligned}
v[x := t] &= t, \texttt{si } v = x \\
v[x := t] &= v, \texttt{si } v \neq x \\
c[x := t] &= c, \texttt{con } c \texttt{ una constante} \\
f(t_1, \ldots, t_k)[x := t] &= f(t_1[x := t], \ldots, t_k[x := t])
\end{aligned}
$$

De manera similar, la definición de sustitución sobre fórmulas se muestra a continuación [9].

$$
\begin{aligned}
\top[x := t] &= \top \\
\bot[x := t] &= \bot \\
P(t_1, \ldots, t_k)[x := t] &= P(t_1[x := t], \ldots, t_k[x := t]), \texttt{con P un predicado} \\
\neg\varphi[x := t] &= \neg(\varphi[x := t]) \\
(\varphi \oplus \psi)[x := t] &= (\varphi[x := t]) \oplus (\psi[x := t]), \texttt{con } \oplus \texttt{ un conectivo lógico} \\
(\forall v\ \varphi)[x := t] &= \forall v(\varphi[x := t]), \texttt{si } v \notin \{x\} \cup vars(t) \\
(\exists v\ \varphi)[x := t] &= \exists v(\varphi[x := t]), \texttt{si } v \notin \{x\} \cup vars(t)
\end{aligned}
$$

En donde `vars(t)` es una función que devuelve el conjunto de variables que figuran en el término `t`.

Nótese que en la definición anterior los casos de los cuantificadores son aplicaciones parciales. Esto sucede porque algunas sustituciones no son válidas, pues pueden dar como resultado nuevas variables ligadas dentro un cuantificador existente. Para resolver este problema se debe especificar cuándo las sustituciones son válidas.

En Haskell, la sustitución se modela de la siguiente forma.

```
type Sustitucion = (String, Term)
```

Código 4.3: Definición del tipo de dato `Sustitucion`.

Un concepto de importancia que será de ayuda para temas posteriores, es la composición de sustituciones, que básicamente se refiere a aplicar una sustitución a otra. Dados dos

conjuntos de sustituciones $\sigma_1 = \{x_1 := t_1, \ldots, x_k := t_k\}$ y $\sigma_2 = \{y_1 := s_1, \ldots, y_n := s_n\}$, la composición $\sigma_1\sigma_2$ se obtiene de eliminar del conjunto:

$$\{x_1 := t_1\sigma_2, \ldots, x_k := t_k\sigma_2, y_1 := s_1, \ldots, y_n := s_n\}$$

las sustituciones de la forma $x := x$ y las sustituciones $y_i := s_i$ tales que $y_i \in \{x_1, \ldots, x_k\}$

Por ejemplo, dados $\sigma_1 = \{x := z, y := f(x, w)\}$ y $\sigma_2 = \{x := r\}$, se tiene:

- $\sigma_1\sigma_2 = \{x := z, y := f(r, w)\}$
- $\sigma_2\sigma_1 = \{x := r, y := f(x, w)\}$

4.2.3. Relación de α-equivalencia

Dadas las restricciones de la función anterior, la sustitución es una función parcial. Sin embargo, nótese que el cambiar el nombre de las variables ligadas de una fórmula no altera el significado de ésta, es decir, las fórmulas $\forall x P(x)$ y $\forall y P(y)$ tienen el mismo significado a pesar de la diferencia en el nombre de la variable.

Se dice que dos fórmulas son *α-equivalentes* (denotado como $\varphi \backsim_\alpha \psi$) si y solo si difieren a lo más en los nombres de las variables ligadas [9].

De manera que, usando la relación de α-equivalencia, es posible hacer que la sustitución sea una función total. Si es el caso que no se puede aplicar una sustitución a una fórmula entonces se obtiene una fórmula α-equivalente a la cual se le puede aplicar ésta, como se muestra a continuación.

$$\forall x(P(x, y))[y := f(x)] \backsim_\alpha \forall z(P(z, y))[y := f(x)] = \forall z(P(z, f(x)))$$

4.2.4. Algoritmo de Unificación

En la Lógica de Primer Orden, así como en la Lógica Proposicional, una manera de determinar si un argumento es válido, es mediante la regla de resolución binaria.

Sin embargo, para entender como funciona esta regla en la Lógica de Primer Orden primero se debe comprender el concepto de *unificación*.

La *unificación* es el proceso de determinar si dos expresiones pueden ser *unificadas*, es decir, hacerlas idénticas aplicando sustituciones apropiadas a sus términos [6]. A una sustitución σ se le conoce como *unificador* de dos expresiones φ_1, φ_2 si $\sigma(\varphi_1) = \sigma(\varphi_2)$ [6].

Un unificador σ de un conjunto de términos W, se llama *unificador más general* (umg) si para cada unificador τ de W, existe una sustitución ρ tal que $\sigma\rho = \tau$, esto es, si dada una lista de unificadores de un conjunto W, se toma uno de los unificadores σ y al componerse con el resto se obtiene exactamente la misma sustitución, entonces σ es un umg [9].

Si dos términos son unificables, debe existir el unificador más general (UMG) que es tan general o más general que cualquier otro unificador [6]. El objetivo de los algoritmos de unificación es encontrar el UMG, si existe. La importancia de los unificadores más generales es que nos permiten representar de manera finita un número infinito de sustituciones [9].

Existen diversos algoritmos de unificación, como el Algoritmo de Robinson o el Algoritmo Martelli-Montanari, el cual se revisa en este capítulo.

El Algoritmo Martelli-Montanari opera sobre un conjunto W de ecuaciones de la forma $\{s_1 = r_1, \ldots, s_k = r_k\}$, y una sustitución σ inicialmente vacía, teniendo como objetivo el unificar de manera simultanea s_i con r_i y generando como resultado un unificador más general para W.

El algoritmo empieza eligiendo una ecuación de forma arbitraria y según sea la forma de ésta se realiza la acción asociada, como se muestra a continuación [12].

1)	x = x	eliminar la ecuación
2)	$f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$	reemplazar por las ecuaciones $s_1 = t_1, \ldots, s_n = t_n$
3)	$f(s_1, \ldots, s_n) = g(t_1, \ldots, t_n)$	falla
4)	$P(s_1, \ldots, s_n) = P(t_1, \ldots, t_n)$	reemplazar por las ecuaciones $s_1 = t_1, \ldots, s_n = t_n$
5)	x = t o t = x	Agregar $[x := t]$ a σ, aplicar la sustitución $[x := t]$ a W y a los terminos en σ
6)	$x = t$ o $t = x$ donde x figura en t y $x \neq t$	falla

El algoritmo termina con éxito cuando W esta vacío y siendo σ el UMG. En caso contrario, W no es unificable.

4.2.5. Resolución Binaria

Habiendo revisado el concepto de unificación, ahora se procede a estudiar la resolución binaria en la Lógica de Primer Orden. La regla de resolución binaria trabaja con formulas en forma clausular. Se recordará que toda fórmula puede transformarse a forma clausular donde todas las cláusulas tienen variables ajenas y todas las variables se consideran cuantificadas universalmente, aunque los cuantificadores no se escriben [9].

Supongase que se tienen dos clausulas ψ y φ de la forma $P \vee \ell$ y $Q \vee \ell^c$ respectivamente, la regla de resolución binaria se define como sigue [9]:

$$\frac{Var(\psi) \cap Var(\varphi) = \emptyset \quad \sigma = umg(\ell, \ell^c)}{(P \vee Q)\sigma}$$

4.3. Desarrollo de la práctica

Nota

Con la finalidad de simplificar los ejemplos de los ejercicios mostrados en esta sección, se opta por utilizar fórmulas proposicionales en notación matemática en lugar de representalas usando el tipo de dato `Formula`. Es responsabilidad del alumno hacer la traducción correspondiente.

1. Definir la función `libres :: Formula -> [String]` que devuelve el conjunto de variables libres de una fórmula, es decir, aquellas que no están ligadas.

```
ghci> libres ∀x(P(x,y) ∧ ∃y(Q(y,z)))
["y", "z"]
```

2. Definir la función `ligadas :: Formula -> [String]` que devuelve el conjunto de variables ligadas de una fórmula.

```
ghci> ligadas ∀x(P(x,y) ∧ ∃y(Q(y,z)))
["x", "y"]
```

3. Definir la función `sustv1 :: Formula -> Sustitucion -> Formula` que trata de aplicar una sustitución a una fórmula. Si no es posible aplicarla entonces se debe devolver un error explicativo.

```
ghci> sustv1 ∀x(P(x,y) ∧ ∃y(Q(y,z))) ("z", Var "y")
***Exception: No se puede sustituir a una variable libre por una
   ligada
ghci> sustv1 ∀x(P(x,y) ∧ ∃y(Q(y,z))) ("z", Var "w")
∀x(P(x,y) ∧ ∃y(Q(y,w)))
```

4. Definir la función `sonAlfaEquiv :: Formula -> Formula -> Bool` que determina si dos for-

mulas son alfaequivalentes.

Sugerencia: Dos formulas son α-equivalentes si una puede ser transformada en la otra cambiando solo las variables ligadas.

```
ghci> sonAlfaEquiv ∀xP(x,y) ∨ ∃yQ(x,y,z) ∀wP(w,y) ∨ ∃rQ(x,r,z)
True
ghci> sonAlfaEquiv ∀xP(x,y) ∨ ∃yQ(x,y,z) ∀wP(w,y) ∨ ∃zQ(x,z,z)
False
```

5. Definir la función `renombra :: Formula -> Formula` que dada una formula, renombre las variables ligadas de ésta, de manera que la intersección entre el conjunto de variables libres y ligadas sea vacio.

```
ghci> renombra ∀xP(x,y) ∨ ∃yQ(x,y,z)
∀wP(w,y) ∨ ∃rQ(x,r,z)
```

6. Definir la función `sustv2 :: Formula -> Sustitucion -> Formula` que aplique una sustitución a una fórmula, usando α-equivalencia de ser necesario.

```
ghci> sustv2 ∀x(P(x,y) ∧ ∃y(Q(y,z))) ("z", Var "y")
∀w(P(w,y) ∧ ∃r(Q(r,y)))
```

7. Definir la función `unifica :: Formula -> Formula -> [Sustitucion]` que obtenga el umg de dos fórmulas. Si no es posible obtener un umg, se debe regresar un error.

   Nota: Aunque la unificación solo esta definida para terminos, se pretende usar esta función para la implementación de la regla de resolución binaria. Tener en cuenta el caso de los predicados, cuya regla de unificación es análoga al caso de las funciones.

```
ghci> unifica Q(x,y,x) Q(y,g(x),x)
*** Exception: No es posible unificar "y" con "g(x)"
ghci> unifica Q(a,c,f(x)) Q(z,c,v)
[("y", Fun "c" []), ("z",Fun "a" []), ("v", Fun "f" [Var "x"])]
```

8. Definir la función `hayResolvente :: Formula -> Formula -> Bool` que determine si es posible obtener un resolvente a partir de dos formulas.

```
ghci> hayResolvente Q(x,y,x) Q(y,g(x),x)
False
ghci> hayResolvente ¬P(x) ∨ Q(f(x)) P(y) ∨ ¬R(f(x))
True
```

9. Definir la función `resolucion :: [Formula] -> [Formula] - [Formula]` que dadas dos claúsulas, obtenga el resolvente obtenido de aplicar la regla de resolución binaria.

```
ghci> resolucion [Q(x,y,x)] [¬Q(y,g(x),x)]
[[]] -- Se obtiene la claúsula vacía
ghci> resolucion [P(x),Q(f(x))] [P(z),¬Q(f(y))]
[P(x),P(z)]
```

Parte II

Programación Lógica

Práctica 5

Introducción a la Programación Lógica con Prolog

5.1. Objetivos

- Introducir al alumno a la programación lógica usando el lenguaje de programación Prolog.
- Familiarizar al alumno con algunos elementos básicos de Prolog.

5.2. Introducción

5.2.1. Prolog

Prolog es un lenguaje de programación lógico que pertenece al estilo declarativo. Entre los principales pilares que sustentan su funcionamiento se encuentra la Lógica de Predicados, estudiada en el capítulo anterior, así como la unificación de términos y el *backtracking*[1].

El enfoque de Prolog se trata más de describir hechos conocidos y relaciones entre objetos sobre un problema, y menos de escribir la secuencia de pasos que se necesitan para resolver el problema. De esta manera, por medio de secuencias lógicas Prolog logra alcanzar una conclusión lógica partiendo de predicados determinados.

[1] En español *retroceso*.

5.2.2. Instalación de SWI Prolog

Antes de comenzar a escribir programas en cualquier lenguaje de programación, es necesario instalar, ya sea un compilador o un interprete. Para el caso de Prolog, existen diversos compiladores, uno de los más conocidos y el cual se usa en este capítulo es `SWI-Prolog`.

Para el caso de Windows y MacOS existen archivos binarios que se pueden descargar en https://www.swi-prolog.org/download/stable. En el caso de Linux, los detalles de la instalación se encuentran en https://www.swi-prolog.org/build/unix.html.

Una vez concluida la instalación, se puede verificar que ésta haya sido exitosa ejecutando el comando `swipl -version`, lo que mostrará la versión instalada.

SWI-Prolog también funciona como un ambiente interactivo, el cual puede iniciarse ejecutando el comando `swipl`, lo que mostrará un mensaje de bienvenida seguido del indicador de una *meta*[2].

```
Welcome to SWI-Prolog (threaded, 64 bits, version 8.4.3)
?-
```

Entre los comandos más utiles para interactuar con el ambiente se tienen:

- `help(X).`, que sirve para mostrar más información acerca de X.
- `halt.`, detiene la ejecución de SWI-Prolog.
- `[archivo1, archivo2, ...].` ,que sirve para cargar las definiciones de archivos y poder usarlas en el ambiente. Alternativamente, también puede cargar el programa pasando el nombre del archivo como argumento a SWI-Prolog: `swipl -s program.pl.`
- `make.` para actualizar las definiciones de algún archivo previamente cargado en el ambiente.

Es importante destacar que el punto al final de cada comando es importante, de esta manera

[2]La definición de meta se encuentra en la sección 5.2.4.

PROLOG sabe que ahí termina el comando.

5.2.3. Primitivas de PROLOG

En PROLOG solo existe un tipo de dato: el término. Sin embargo éste se divide en 4 subtipos:

Átomos

Existen dos tipos de átomos, aquellos que son formados mediante el uso de letras y números, y aquellos que se forman usando signos unicamente [4]. El primer tipo normalmente comienza con una letra minúscula. Por otro lado, los átomos formados por signos usualmente solo contienen signos. Puede ser el caso que se deba definir un átomo que comience con una letra mayúscula o un dígito. Para que éste sea un átomo valido, se debe delimitar con comillas simples, y entonces se puede definir un átomo usando cualquier carácter en su nombre. Como ejemplos de átomos se tienen:

```
abc logica alan_turing 'Robert Kowalski' prolog1972
```

Números

Como parte de las primitivas del lenguaje, se tienen números enteros y flotantes.

```
17 -2.62e2 0 1 3.33 -512 0.001 3.1415
```

Variables

Las variables son muy similares a los átomos, con la diferencia que estas empiezan con una letra mayúscula o con un guión bajo. Las variables representan algún objeto que no se conoce o no se quiere nombrar en el momento en que se escribe el programa.

Existe una variable especial que se llama *variable anónima*, la cual se representa con el símbolo `_`. Las variables anónimas son útiles cuando se quiere hacer uso de una variable, pero no se necesita su valor. Este tipo de variables usualmente se usan en el cuerpo de una regla, que se estudiará más adelante.

```
X Y Z Respuesta Nombre _
```

Términos compuestos

Los términos compuestos se escriben como `f(a1, a2, ..., an)`, donde `f` es un átomo conocido como *funtor* y `a1, a2, ..., an` son los argumentos. Los argumentos pueden ser cualquier tipo de término, incluyendo otros términos compuestos.

```
f(a) g(b) f(g(a,b)) comidaFavorita(fernando, tacos)
```

5.2.4. Definición de hechos, reglas y consultas

Construir un programa en PROLOG se resume en

- Establecer *hechos* sobre objetos y sus relaciones.
- Definir *reglas* sobre objetos y sus relaciones.
- Hacer *consultas* sobre objetos y sus relaciones.

Al conjunto de hechos y reglas se le conoce como *base de conocimientos*.

Hechos

Un hecho se define como una relación explícita entre objetos y las propiedades que estos objetos pueden tener [4].

Un hecho debe comenzar con un predicado, que es un átomo, y terminar con un punto.

El predicado puede ir seguido de uno o más argumentos entre paréntesis. Los argumentos pueden ser átomos (en este caso, estos átomos se tratan como constantes), números, variables o listas.

En un programa escrito en PROLOG, la presencia de un hecho indica que una afirmación es verdadera mientras que la ausencia de un hecho indica una afirmación que es falsa. Los siguientes son ejemplos de hechos.

```
logica.
algoritmos.
fernando.
alumno(fernando).
alumno(luis).
calificacion(fernando, algoritmos, 9).
calificacion(luis, probabilidad, 4).
```

Reglas

Una regla puede verse como una extensión de un hecho con condiciones adicionales que deben cumplirse para que ésta sea verdadera [4].

En PROLOG una regla se conforma de una cabeza y un cuerpo, que son conectados por el símbolo `:-`. La cabeza es similar a un hecho (un predicado con argumentos), mientras que el cuerpo es una secuencia de predicados separados por comas.

La sintaxis de una regla es la siguiente:

$$\texttt{term :- term}_1\texttt{, term}_2\texttt{, ..., term}_k$$

Lo anterior se traduce en que si se quiere que *term* sea verdadero, entonces $term_1$, $term_2$, ... $term_k$ deben ser verdaderos.

Por ejemplo, para definir que alguien aprobó un examen se tendría que cumplir que esa persona haya presentado el examen de la materia y haya obtenido una calificación mayor a 6, lo que se podría traducir a PROLOG como:

```
aprobo(Persona, Materia) :- alumno(Persona), calificacion(Persona,
    Materia, Nota), Nota >= 6.
```

Nótese que algunas variables aparecen mulitples veces, por ejemplo, la variable `Persona` aparece 3 veces. Cada vez que una variable se instancia en algún término, todas las apariciones de ella se instancian dentro del alcance de ésta. En particular, en las reglas el alcance de una variable es toda la regla. Por ejemplo, en la regla `aprobo`, si sucede que `Persona` se instancia a `fernando`, entonces PROLOG tratará de encontrar valores para `Materia` y `Nota` que hagan que tanto `calificacion(fernando, Materia, Nota)` como `Nota >= 6` sean verdaderos.

Metas y consultas

Una vez que se tienen hechos y reglas definidas, es posible empezar a hacer consultas o buscar metas. Cuando se ejecuta un intérprete de Prolog, se puede observar el símbolo `?-`, que sirve para ingresar una meta o una consulta.

Una *meta* es una declaración que comienza con un predicado y va seguida de sus argumentos, los cuales deben ser constantes. El predicado de una meta válida debe haber aparecido en al menos un hecho o regla en el base de conocimientos, y el número de argumentos en la meta debe ser el mismo que aparece en el base de conocimientos.

```
?- calificacion(fernando, probabilidad, 10).
false.
?- calificacion(fernando, algoritmos, 9).
true.
?- aprobo(luis, probabilidad).
false.
```

Una *consulta* es una declaración que comienza con un predicado y va seguida de sus argumentos, algunos de los cuales pueden ser variables. De manera similar a las metas, el predicado de una consulta válida debe haber aparecido en al menos un hecho o regla en la base de conocimientos, y el número de argumentos en la consulta debe ser el mismo que

aparece en la base de conocimientos.

```
?- calificacion(luis, Materia, Calificacion).
Materia = probabilidad,
Calificacion = 4.
?- aprobo(Persona, algoritmos).
Persona = fernando.
```

5.2.5. Operadores

Algunas veces, principalmente por motivos de legibilidad, es conveniente escribir algunos functores como operadores, esto es, de manera infija. Por ejemplo, para el caso de la aritmetica, se recomienda el uso de operadores, ya que escribir `x+y*z` sin el uso de operadores se vería como `+(x,*(y,z))`, que a pesar de ser un término bien formado, es complicado de leer.

Entre los operadores aritmeticos más comunes se encuentran:

Nombre	Operador
Suma	`+`
Resta	`-`
Multiplicación	`*`
División real	`/`
División entera	`//`
Potencia	`**`
Módulo	`mod`

```
?- X = 2+2*3.
?- X = 2**5.
?- X = 5 * 2.
```

Código 5.1: Ejemplos de uso de operadores aritméticos

Obsérvese que al tratar de satisfacer la meta `X = 1+2.` el resultado no es el esperado.

```
?- X = 1+2.
X = 1+2.
```

Esto sucede porque `1+2` es un término y no una expresión aritmética como tal. Para poder obtener el resultado de una expresión, PROLOG provee el operador `is`. El segundo argumento de este operador es un término que se interpreta como una expresión aritmética. Para que un `is` tenga éxito, se obtiene el resultado del segundo argumento y se trata de unificar con el primer argumento.

```
?- X is 1+2.
X = 3.
```

Código 5.2: Ejemplo de uso de `is`

Por otro lado, también se cuenta con operadores de relación:

Nombre	Operador
Igualdad	`=:=`
Desigualdad	`=\=`
Menor que	`<`
Mayor que	`>`
Menor o igual	`=<`
Mayor o igual	`>=`

```
?- 2 > 3.
?- 1 =\= 1.
?- 10 =:= 10.
```

Código 5.3: Ejemplos de uso de operadores de relación

En otros lenguajes de programación el símbolo de igualdad se usa para *asignar* valores. En PROLOG éste es conocido como el operador de igualdad (`=`). Su objetivo es tratar de unificar dos términos. De esta manera el término `X = 1` no está especificando que a la variable `X` se le esta asignando el valor 1, sino que se quiere que la variable `X` se unifique con el término `1`.

```
?- prolog = X.
X = prolog.
?- 2012 = 2012.
true.
?- termino = termino.
true.
```

5.2.6. Manejo de listas

Un recurso muy importante a la hora de programar es el uso de estructuras de datos. Así como la mayoría de los lenguajes de programación, PROLOG cuenta con listas como principal estructura de datos.

Una lista es una secuencia de elementos. En PROLOG las listas son heterogeneas, es decir, el tipo de los elementos de una lista no debe ser necesariamente el mismo para todos los elementos. Una lista es una lista vacía (sin elementos) o es una estructura que tiene dos componentes: la cabeza y la cola. La lista vacía se escribe como `[]`. Por otro lado la cabeza y la cola de una lista son argumentos del operador |.

Los siguientes son ejemplos de listas en PROLOG:

```
[].
[1 | []]. % [1]
[1|[2|[]]]. % [1,2]
```

Se puede notar que el uso del operador `|` no parece ser muy conveniente a la hora de definir listas y es por eso que usualmente no se usa. En cambio, se usa la notación de listas, que consiste en escribir los elementos de una lista separados por comas y encerrados entre corchetes.

```
[].
[a].
[1,2,3].
```

Aunque es preferible usar la notación de listas, el uso del operador | es de utilidad a la hora de definir predicados sobre listas, ya que permite usar el patrón `[X|XS]` para denotar que `X` es la cabeza de la lista y `XS` es la cola de la lista.

```
primero([X|XS], X).
```

Código 5.4: Definición del predicado *primero* para listas

Entre los predicados más útiles a la hora de trabajar con listas se encuentra:

- `member(X, XS)`, que verifica si un elemento X esta contenido en una lista XS.

```
?- member(1, [1,2,3]).
true.
?- member(a, [1,2,3]).
false.
```

- `append(XS, YS, XXS)`, que unifica en XXS la concatenación de las listas XS y YS.

```
?- append([1,2,3], [4,5], XXS).
XS = [1,2,3,4,5].
```

- `length(XS, L)`, que unifica en L la longitud de la lista XS.

```
?- length([1,2,3], L).
L = 3.
?- length([], L).
L = 0.
```

5.3. Desarrollo de la práctica

1. Considerese la siguiente base de conocimientos en donde figura el predicado `padre` que relaciona a dos personas, donde la primera es padre de la segunda:

```
padre(a,b).
padre(a,c).
padre(b,d).
padre(b,e).
padre(c,f).
```

   Usando solo los hechos de la base de conocimiento y predicados primitivos de PROLOG, define los siguientes predicados:

   - `hermano(X,Y)`, que es verdadero si X y Y son hermanos.
   - `primo(X,Y)`, que es verdadero si X y Y son primos.
   - `nieto(X,Y)`, que es verdadero si Y es nieto de X.
   - `descendiente(X,Y)`, que es verdadero si Y es descendiente de X.

2. El Dr. Vegapunk ha estado trabajando en la cura un virus mortal de manera secreta. Después de varios meses ha logrado crear de manera satisfactoria una cura. Sin embargo, cada experimento ha sido guardado en frascos de colores diferentes y ahora no sabe en cual frasco se encuentra la vacuna final. El Dr. Vegapunk te ha contactado para que le ayudes a saber en qué frascos se encuentra la cura.

   Lo que se sabe es que cada frasco tiene un color diferente: rojo, anaranjado, amarillo, verde, azul y violeta. Después de cierto tiempo el doctor recordó que hay tres pares de frascos, en los que para cada par, sabemos que en uno de los frascos está la cura:

   - los frascos violeta y azul.
   - los frascos rojo y amarillo.
   - los frascos azul y anaranjado.

Por otro lado, el doctor también recuerda que hay tres pares donde para cada par, hay uno en el que definitivamente no está la vacuna.

- el violeta y el amarillo.
- el rojo y el anaranjado.
- el verde y el azul

Además, parece ser que el frasco rojo contiene mermelada de fresa.

Define, usando esta información, las consultas y base de conocimientos necesarias para ayudar al Dr. Vegapunk a encontrar la cura.

```
?- cura(rojo)
false.
?- cura(azul)
???
```

3. Define el predicado `nth(XS, N, E)` que es verdadero si E es el n-ésimo elemento de XS.

```
?- nth([1,2,3], 1, E).
E = 1
?- nth([1,2,3], 10, E).
false.
```

4. Define el predicado `rota(XS,N,L)` que es verdadero si L es la lista resultante de rotar N veces los elementos de XS a la derecha.

```
?- rota ([1 ,2 ,3 ,4] , 2 , X ) .
X = [3 ,4 ,1 ,2]

?- rota ([1 ,2 ,3 ,4] , 0 , X ) .
X = [1 ,2 ,3 ,4]
```

5. Define el predicado `agrupa(XS,XXS)` que es verdadero si XXS es la lista de listas resultante de agrupar los elementos consecutivos iguales de XS.

```
?- agrupa ([1 ,1 ,1 ,2 ,2 ,3 ,3 ,3 ,4 ,4 ,1 ,1 ,5] , Z ) .
Z = [[1 ,1 ,1] ,[2 ,2] ,[3 ,3 ,3] ,[4 ,4] ,[1 ,1] ,[5]]

?- agrupa ([1 ,2 ,3 ,4 ,5 ,6] , Z ) .
Z = [[1] ,[2] ,[3] ,[4] ,[5] ,[6]]
```

Práctica 6

Control de programas en PROLOG

6.1. Objetivos

- Introducir algunos mecanismos de PROLOG que permiten modificar la búsqueda de soluciones.
- Utilizar dichos mecanismos con el fin de hacer más eficientes los programas escritos en PROLOG.

6.2. Introducción

En 1979 Robert Kowalski publicó un artículo con el título *Algorithm = Logic + Control* [16], en donde establecía lo siguiente:

"*Se puede considerar que un algoritmo esta formado por un componente lógico, que especifica el conocimiento a ser utilizado en la resolución del problema, y un componente de control, que determina las estrategias de resolución de problemas para el cual se utiliza ese conocimiento. El componente lógico determina el significado del algoritmo mientras que el componente de control sólo afecta a su eficiencia. La eficiencia de un algoritmo a menudo puede ser mejorada al mejorar el componente de control sin cambiar la lógica del algoritmo.*"

En PROLOG el componente lógico se refiere a los hechos, reglas y consultas, mientras que el componente de control se refiere a los mecanismos usados para obtener la respuesta a una consulta, como la resolución y el *bakctracking*.

6.2.1. Control de búsqueda

Uno de los pilares sobre el cual PROLOG basa su proceso de búsqueda de soluciones es el algoritmo de *backtracking*. Este algoritmo permite realizar una búsqueda en profundidad de las soluciones de un problema, de modo que si un camino no lleva a una solución, entonces se retrocede y se prueba con otro camino.

A grandes rasgos, esto es lo que ocurre cuando se desea satisfacer una meta:

1. Se inicia una búsqueda en la base de conocimientos, empezando desde el inicio del programa, con el fin de encontrar un hecho o regla que se pueda unificar con la meta.
2. Si la unificación fue exitosa, las variables de la meta se instancian con los valores de las variables de la regla o hecho con el cual se unificó. Aquí se crea lo que se denomina como *punto de elección*, el cual se utiliza para marcar el punto en el que se debe reanudar la búsqueda en caso de que la meta no se pueda satisfacer.
3. Si se ha unificado con una regla, además de la meta inicial, cada predicado que aparece en el cuerpo de la regla se convierte en una nueva meta adicional que debe ser satisfecha.
4. Si no es posible satisfacer la meta, entonces se debe *retroceder*, es decir, se debe volver al último punto de elección y continuar la búsqueda desde ahí. Toda instancia de variable que se haya hecho después del punto de elección se debe deshacer.
5. Se repiten los pasos 1 a 4 hasta que se encuentre una solución o se agoten todas las posibilidades.

Para ilustrar lo anterior, observe la siguiente base de conocimientos:

```
inscribir(fernando, probabilidad)
inscribir(fernando, logica)

% Luis siempre inscribe las mismas materias que Fernando.
% La única materia que no quiere inscribir es probabilidad.
inscribir(luis, M) :- inscribir(fernando, M), M \== probabilidad.
```

Para ver en que materias se ha inscrito Luis, basta con ejecutar la consulta `inscribir(luis, X)`.

```
?- inscribir(luis, M).
M = logica.
```

El proceso a partir del cual se encontró que `M = logica` es el siguiente.

- La meta que se desea satisfacer es `inscribir(luis, M)`. PROLOG empieza a leer de arriba hacia abajo la base de conocimientos y encuentra que la regla `inscribir(luis, X)` se puede unificar con la meta.
- La nueva meta ahora es `inscribir(fernando,M)`, pues asi lo indica la regla. Se busca en la base de conocimientos, y primero se encuentra el hecho `inscribir(fernando, probabilidad)`, con el cual se puede unificar la meta actual, por lo tanto, la variable `M` se instancia con el valor de `probabilidad`.
- Se procede a tratar de satisfacer la siguiente meta que es `probabilidad \== probabilidad`, lo cual es falso, pues ambos son los mismos términos. Dado que la meta ha fallado, entonces se retrocede y se busca una alternativa.
- La meta es de nuevo `inscribir(fernando,M)`. Al elegir previamente el hecho `inscribir(fernando, probabilidad)`, se colocó un punto de elección. Como se quiere retomar la búsqueda, esta continua después de dicho punto. PROLOG encuentra el hecho `inscribir(fernando, logica)`. `M` se instancia como `logica`.
- Se continua con la siguiente meta, que es `logica \= probabilidad`, la cual es satisfacible.
- Como ya no hay mas variables por instanciar la búsqueda termina, obteniendo que `M = logica`.

Es importante destacar que el orden en el que se definen las cláusulas en un programa determina el orden en que se obtienen las soluciones, pues de eso depende como se recorren las ramas del árbol de búsqueda de soluciones. Por lo tanto, se recomienda primero definir los hechos y luegos las reglas asociadas a un predicado. En el caso de las reglas, ordenar los predicados del cuerpo de acuerdo al coste computacional que se requiere para satisfacerlos, de modo que los predicados que requieran menos recursos se coloquen primero.

Tip

Si se desean ver los pasos que PROLOG sigue para satisfacer una meta o encontrar el resultado de una consulta se puede ejecutar el predicado `trace.` antes de cualquier consulta o meta.

```
?- trace.
true.
[trace] ?- inscribir(luis, M). % Meta a satisfacer
Call: (10) inscribir(luis, _65704) ? % Se inicia la búsqueda
% Se encuentra una regla, se unifica y se crea un punto de
   elección
Call: (11) inscribir(fernando, _65704) ?
Exit: (11) inscribir(fernando, probabilidad) ?
% Se intenta satisfacer la siguiente meta
Call: (11) probabilidad\=probabilidad ?
% La meta falla
Fail: (11) probabilidad\=probabilidad ?
% Se retrocede al último punto de elección
Redo: (11) inscribir(fernando, _65704) ?
% Se encuentra una alternativa
Exit: (11) inscribir(fernando, logica) ?
% Se intenta satisfacer la siguiente meta
Call: (11) logica\=probabilidad ?
% La meta se satisface
Exit: (11) logica\=probabilidad ?
% Se ha satisfecho la meta inicial
Exit: (10) inscribir(luis, logica) ?

M = logica.
```

Considere la definición del predicado `factorial` que se muestra en el Código 6.1.

```
factorial(N, Fact):- T is N-1, factorial(T, S1), Fact is N*S1.
factorial(0,1).
```

Código 6.1: Definición del predicado factorial

Veáse lo que sucede al consultar `factorial(2,X).`

```
[trace]  ?- fact(2,X).
Call: (10) fact(2, _7056) ?
Call: (11) _8248 is 2+ -1 ?
Exit: (11) 1 is 2+ -1 ?
Call: (11) fact(1, _9758) ?
Call: (12) _10520 is 1+ -1 ?
Exit: (12) 0 is 1+ -1 ?
Call: (12) fact(0, _12030) ?
Call: (13) _12792 is 0+ -1 ?
Exit: (13) -1 is 0+ -1 ?
Call: (13) fact(-1, _14302) ?
```

Eventualmente se obtendra un error debido al limite de llamadas recursivas, pues como se puede observar, PROLOG siempre unificará la meta deseada con la primera regla definida y por tanto nunca se llegará al caso base. Una solución parcial[1] a este error es simplemente reordenar las definiciones, como se muestra en el código 6.2.

```
factorial(0,1).
factorial(N, Fact):- T is N-1, factorial(T, S1), Fact is N*S1.
```

Código 6.2: Definición del predicado factorial

[1]Existe un camino en el árbol de búsqueda que no termina su ejecución.

```
[trace]  ?- factorial(2, N).
Call: (10) factorial(2, _30020) ? creep
Call: (11) _31324 is 2+ -1 ? creep
Exit: (11) 1 is 2+ -1 ? creep
Call: (11) factorial(1, _32946) ? creep
Call: (12) _33764 is 1+ -1 ? creep
Exit: (12) 0 is 1+ -1 ? creep
Call: (12) factorial(0, _35386) ? creep
Exit: (12) factorial(0, 1) ? creep
Call: (12) _32946 is 1*1 ? creep
Exit: (12) 1 is 1*1 ? creep
Exit: (11) factorial(1, 1) ? creep
Call: (11) _30020 is 2*1 ? creep
Exit: (11) 2 is 2*1 ? creep
Exit: (10) factorial(2, 2) ? creep
N = 2 ;
```

6.2.2. Operador de corte

Existen dos situaciones en las que el *backtracking* toma relevancia a la hora de buscar soluciones. La primera es cuando una meta falla y entonces se debe retroceder en busca de alternativas. La otra es proporcionada por el usuario. Una vez que una meta ha sido satisfecha, es posible continuar recorriendo el árbol en busca de mas soluciones. Por ejemplo, tómese en cuenta la siguiente base de conocimientos.

```
color(azul).
color(rojo).
color(amarillo).
```

Al ejecutar la consulta `color(X)`, se verá lo siguiente:

```
X = azul
```

Pero se puede observar que la respuesta no contiene un punto, esto sucede porque es posible seguir buscando soluciones. Para continuar la búsqueda basta con teclear el símbolo `;`.

```
X = azul ;
X = rojo ;
X = amarillo.
```

Ahora, tómese en cuenta el Código 6.2 y la consulta `factorial(2,X)`.

```
?- factorial(2,X).
X = 2
```

Es posible continuar con la búsqueda, pues al momento de tratar de satisfacer la submeta `factorial(0,X)` hay dos posibles alternativas:

- Usar el hecho `factorial(0,1)`, gracias al cual es posible obtener la respuesta deseada.
- Usar la regla con la definición recursiva, la cual nunca término pues no alcanza el caso base en ningún momento.

El caso anterior es un ejemplo de que hay situaciones en las que no es necesario explorar el árbol de búsqueda completo, pues esto conlleva búsquedas innecesarias o soluciones incorrectas.

PROLOG proporciona un predicado predefinido llamado *corte* que se representa mediante el signo de exclamación `(!)`. Su principal función es reducir el espacio de búsqueda podando las ramás de éste, es decir, eliminando caminos que puedan dar con otras soluciones. Éste predicado se satisface siempre y no puede resatisfacerse, además, cuando se encuentra un corte como meta, todos las alternativas pendientes son descartadas, de ahí un intento de volver a satisfacer cualquier meta entre la meta principal y el corte no será posible.

Por tanto, para solucionar el problema del predicado factorial, basta con usar un corte.

Cuando se trata de satisfacer la meta `factorial(0,1)`, se encuentra el corte y las alternativas son descartadas, en especial, aquella que hace que la regla `factorial(N,fact)` se unifique con `factorial(0,1)` y entonces termine en una búsqueda infinita.

```
factorial(0,1) :- !.
factorial(N, Fact):- T is N-1, factorial(T, S1), Fact is N*S1.
```

Código 6.3: Definición del predicado factorial usando corte.

6.2.3. Negación como falla

La negación como falla es un razonamiento que Robert Kowalski describe como "la derivación de conclusiones negativas dada la falta de información"[17].

El concepto de negación lógica en PROLOG es problemático, en el sentido de que el único método que PROLOG puede usar para saber si una proposición es falsa, es tratar de satisfacerla (simplemente buscando una prueba), de manera que si la búsqueda falla, se concluye que es falsa. El problema de esto, es que es posible que PROLOG no tenga la suficiente información, por lo que no podrá probar la proposición. En tal caso, la falsedad de la proposición es solo relativa a la base de conocimientos, lo que se conoce como la suposición del mundo cerrado:

- Todo lo que es verdadero es derivable de hechos y reglas.
- Si una meta no puede ser satisfacible, se asume como falsa.

En PROLOG la negación como falla se representa mediante el predicado `\+`[23].

Tenga en cuenta, que el predicado `\+` no es una negación lógica, sino que significa que no es posible probar algo.

Por ejemplo, considere el predicado `interseccion(S1,S2)` que determina si dos listas tienen una intersección no vacía y el predicado `disjunto(S1,S2)` que determina la intersección de dos listas es vacía.

[3]Es un mnemotécnico para *no demostrable*, donde \ significa *no* y + demostrable. [20]

```
interseccion(S1, S2) :- member(X, S1), member(X, S2).
disjunto(S1, S2) :- \+ interseccion(S1, S2).
```

Ahora considere las siguientes consultas:

```
?- interseccion([a,b,c], [c,d,e]).
true.
?- disjunto([a,b,c], [c,d,e]).
false.
?- interseccion([a,b,c], [d,e,f]).
false.
?- disjunto([a,b,c], [d,e,f]).
true.
?- disjunto([a,b,c], S).
false.
```

Observe como la última consulta devolvió un `false` en lugar de alguna lista que fuera disjunta con `[a,b,c]`. Dado que el predicado `disjunto` esta definido como la negación de `interseccion` y es posible encontrar una lista que intersecte con `[a,b,c]` entonces el resultado de `\+ interseccion([a,b,c], S2)` es falso.

```
[trace]  ?- \+ interseccion([a,b,c], S2).
Call: (11) interseccion([a, b, c], _1192) ?
Call: (12) lists:member(_2516, [a, b, c]) ?
Exit: (12) lists:member(a, [a, b, c]) ?
Call: (12) lists:member(a, _1192) ?
Exit: (12) lists:member(a, [a|_4820]) ?
Exit: (11) interseccion([a, b, c], [a|_4820]) ?
false.
```

6.3. Desarrollo de la práctica

1. Observe la siguiente consulta y su resultado.

```
?- member(1, [1,1,1,1]).
true ;
true ;
true ;
true.
```

   Como se puede observar, no importa si se ha encontrado un elemento, la búsqueda continúa. Define el predicado `pertenece(X, XS)` que es análogo a `member`, con la diferencia de que basta con encontrar la primer ocurrencia de `X` para detener la búsqueda.

2. Define el predicado `elimina_primera(E, XS, YS)` que es verdadero si `YS` es `XS` después de haber eliminado la primera ocurrencia de `E`.

```
?- elimina_primera(1, [1,2,3,1,1], YS).
YS = [2,3,1,1].
```

3. Un árbol binario de búsqueda es un tipo especial de árbol binario que tiene la siguientes características:

   - El subárbol izquierdo de un nodo contiene solo nodos con elementos menores que el elemento de la raíz.
   - El subárbol derecho de un nodo contiene solo nodos con elementos mayores que el elemento de la raíz.
   - El subárbol izquierdo y derecho también son arboles binarios de búsqueda.

   En Prolog podemos representar árboles binarios de la siguiente manera:

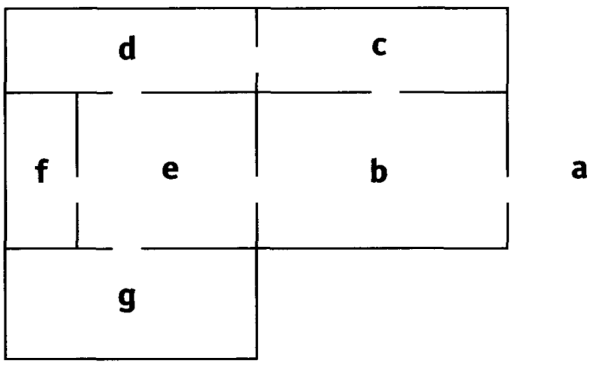
   - El átomo `vacio` representa al árbol vacio.
   - El término compuesto `nodo(N, L, R)` representa un nodo de un árbol en donde

`N` representa el valor de la raíz y `L` y `R` son árboles binarios.

Define el predicado `busca(X, Arbol)` que es verdadero si `X` es un elemento de `Arbol`. **Nota:** Asegurarse de no hacer búsquedas innecesarias. Para eso se debe de usar el operador de corte.

```
?- busca(2, nodo(2, nodo(1, vacio, vacio), nodo(3, vacio,
   vacio)).
true.
?- busca(10, nodo(2, nodo(1, vacio, vacio), nodo(3, vacio,
   vacio)).
false.
```

4. Tómese en cuenta la siguiente base de conocimientos que representa el plano de una casa, especificando entre que habitaciones hay una puerta, como se muestra en el dibujo:




```
puerta(a,b).
puerta(b,e).
puerta(b,c).
puerta(d,e).
puerta(c,d).
puerta(e,f).
puerta(g,e).
```

- Define el predicado `camino(X,Y,P)` que es verdadero si `P` es un camino que va de

X a Y. Asegúrese de no repetir cuartos.

```
?- camino(a,d, P).
P = [a,b,c,d] ;
P = [a,b,e,d].
```

- Define el predicado `primer_camino(X,Y,P)` que es análogo al predicado anterior con la diferencia de que solo se debe encontrar un solo camino, es decir, no debe ser posible encontrar soluciones alternas.

```
?- primer_camino(a,d, P).
P = [a,b,c,d].
```

Parte III

Razonamiento Ecuacional y Deducción Natural

Práctica 7

Introducción al asistente de pruebas Coq

7.1. Objetivos

- Familiarizar al alumno con algunos elementos básicos del asistente de pruebas Coq.
- Realizar algunas demostraciones sobre la Lógica Proposicional haciendo uso de Coq.

7.2. Introducción

La escalabilidad y la complejidad de los sistemas de cómputo modernos, la cantidad de personas involucradas y la variedad de requirimientos que se imponen a estos, hacen que sea un desafío crear software que sea lo más confiable posible, es decir, que contenga la menor cantidad de errores y brechas de seguridad posibles.

Los científicos de la computación y los ingenieros de software han tratado de responder a estos desafíos mediante el desarrollo de una serie de técnicas para mejorar la confiabilidad del software, que van desde recomendaciones sobre la gestión de proyectos de software, hasta filosofías de diseño para bibliotecas (por ejemplo, modelo-vista-controlador, pub-sub, entre otras) y lenguajes de programación, a técnicas matemáticas para especificar y razonar sobre las propiedades del software y herramientas para ayudar a verificar estas propiedades [27]. Esta sección se centra en éstas últimas.

Existen diferentes herramientas que permiten verificar propiedades de programas, entre ellas

se encuentran los *asistentes de pruebas*[1], que son herramientas que automatizan los aspectos más rutinarios de la construcción de pruebas mientras que dependen de la guía humana para los aspectos más difíciles. En este capítulo se introduce al asistente de pruebas Coq, que es un asistente de pruebas basado en el cálculo de construcciones inductivas (CCI) [2].

El asistente de pruebas Coq es un ambiente que permite definir objetos (enteros, conjuntos, árboles, funciones, programas, etc); hacer declaraciones sobre éstos usando predicados básicos y conectores lógicos, y finalmente escribir pruebas para verificar que dichas definiciones cumplen ciertas especificaciones.

7.2.1. Instalación

El sitio web https://coq.inria.fr/download contiene las instrucciones detalladas para instalar Coq en los sistemas operativos MacOS y Windows, así como para diferentes distribuciones de Linux.

Si bien es posible usar Coq mediante `coqtop`, que es un comando que se ejecuta mediante la terminal, se recomienda instalar `CoqIDE`, que es un ambiente gráfico.

Nota

Por motivos ilustrativos, en este capítulo se usará `coqtop` para ejecutar los comandos de Coq, sin embargo, se recomienda usar `CoqIDE` para realizar las actividades.

[1] En inglés *proof assistant*.

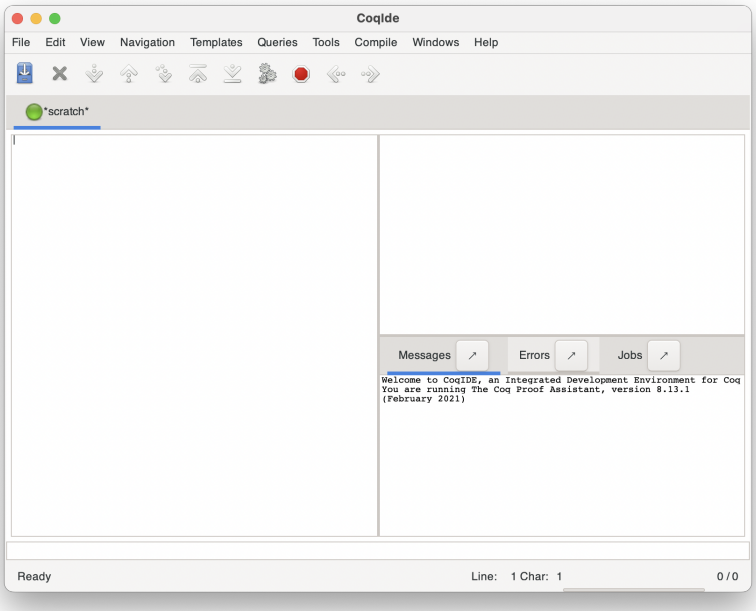


Figura 7.1: CoqIDE

7.2.2. Términos

Per se, Coq no es un lenguaje de programación. Sin embargo, uno de sus multiples componentes es Gallina, que es el lenguaje de especificación de Coq, el cual permite representar los tipos y programas de los lenguajes de programación habituales.

Nota

Coq esta escrito en Ocaml, que es un lenguaje de programación funcional con una sintaxis muy similar a la de Haskell. Por tanto conceptos como la aplicación de funciones, uso de operadores, aritmética, condicionales, entre otros[2], son omitidos en esta sección.

En Coq, todas las expresiones son términos, y todos los términos tienen un tipo. Hay tipos para funciones, hay tipos atómicos, y también tipos para demostraciones e incluso tipos para

[2]Si se desea profundizar en estos temas, se puede consultar la sección *Functional Programming in Coq* de la referencia [27].

los tipos mismos. La expresión *x tiene el tipo T* se escribe como `x:T`.

Tip

`Check` es un comando que recibe un término como argumento y verifica que sea un término bien formado, además de mostrar el tipo de éste. Es importante mencionar que los comandos siempre terminan con un punto.

```
(* nat es un tipo predefinido para los números naturales *)
Coq < Check nat.
nat : Set
(* 0 es un número natural *)
Coq < Check 0.
0 : nat
(* S es la función sucesor *)
Coq < Check S.
S : nat → nat
(* plus es una función binaria sobre los números naturales *)
Coq < Check plus.
Nat.add : nat → nat → nat
```

7.2.3. Tipos básicos

Coq cuenta con una serie de tipos de datos predefinidos, los cuales pueden ser consultados en la biblioteca `Coq.Init.Datatypes`, entre ellos se encuentran los boleanos y los números naturales. Estos últimos pueden ser representados usando la notación de Peano (0, S(0), ...) o usando notación decimal (0,1,2, ...).

Tipo	Ejemplo
nat	0, S(0), S(S(0)), 1, 2
Bool	True, False

7.2.4. Funciones

La sintaxis para definir funciones en Coq es la siguiente:

$$\texttt{Definition}\ \textit{nombre}\ (x_1 : t_1)(x_2 : t_2)\ldots : t := \textit{cuerpo}$$

en donde `Definition` es una palabra reservada de Coq, *nombre* es el nombre de la función seguida de una serie de argumentos, que puede ser vacía, con sus respectivos tipos, `t` es el tipo que devuelve la función y *cuerpo* el cuerpo de la función.

Por ejemplo, la definición de la función `cuadrado` que calcula el cuadrado de un número se ve como en el Código 7.1

```
Definition cuadrado (x:nat) : nat := x * x.
```

Código 7.1: Definición de la función `cuadrado` en Coq

Tip

El comando `Compute` sirve para evaluar términos. Es de gran ayuda para verificar que las funciones esten correctamente definidas.

```
Compute (cuadrado 2).
4 : nat.
```

7.2.5. Proposiciones

Como se recordará, una proposición es una declaración sobre la cual se puede decir si es verdadera o falsa. En Coq, las proposiciones también son términos y tienen el tipo `Prop`.

En la Figura 7.2 se muestran los elementos sintácticos de las proposiciones, en el primer renglón se muestran los elementos lógicos y en el segundo la notación correspondiente en Coq.

$\bot$	$\top$	$t = u$	$t \neq u$	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \to Q$	$P \Leftrightarrow Q$
`False`	`True`	`t=u`	`t<>u`	`~P`	`P /\ Q`	`P \/ Q`	`P -> Q`	`P <-> Q`

Figura 7.2: Elementos sintácticos de las proposiciones en Coq.

También es posible hacer uso de cuantificadores.
Para el caso del cuantificador universal ($\forall$) se utiliza la notación:

`forall` $(x_i : t_i) \ldots$, `P`

En donde P es una proposición y x_i son variables de tipo t_i que son ligadas por el cuantificador.

Para el caso del cuantificador existencial ($\exists$) se utiliza la notación:

`exists` $(x : t)$, `P`

En donde P es una proposición y x es una variable de tipo t que es ligada por el cuantificador.

```
Coq < Check forall (B : Prop) (A : Prop), A ∧ B → B ∨ B.
forall A B : Prop, A ∧ B → B ∨ B : Prop

Coq < Check exists (A : Prop), A ∧ A.
exists A : Prop, A ∧ A : Prop
```

7.2.6. Negación

Coq se basa en la lógica intuicionista, por tanto no hay reglas para la negación ($\neg$) ni para lo falso ($\bot$) [9]. Por tanto, la negación de una proposición P se define como $P \to \bot$.

```
Coq < Print not.
not = fun A : Prop ⇒ A → False : Prop → Prop
```

El carácter constructivo de la negación restringe a la lógica de una manera importante. Proposiciones que son válidas en la lógica clásica, como la ley del tercero excluido ($P \vee \neg P$), no son válidas en la lógica intuicionista [9].

7.2.7. Lógica clásica

Aunque Coq se basa en la lógica intuicionista, es posible utilizar la lógica clásica mediante la biblioteca Classical. Esta biblioteca, entre otras cosas, define el axioma de la ley del tercero excluido ($P \vee \neg P$).

```
Coq < Require Import Classical.
Coq < Check classic.
classic : forall P : Prop, P ∨ ¬P
```

7.2.8. Tácticas y Demostraciones

Existen varias formas para determinar si una proposición es verdadera o falsa. Una de ellas es creando una prueba. En Coq esto se logra haciendo un razonamiento hacia atrás con tácticas. Una táctica transforma una meta en un conjunto de submetas de tal manera que con resolver estas submetas es suficiente para resolver la meta original. La prueba tiene éxito cuando no quedan metas secundarias. [26]

Para crear una prueba primero debe de haber una meta, la cual se puede crear haciendo uso de alguno de los siguientes comandos, en donde *prop* representa una proposición lógica.

`Lemma` *nombre* `: prop` `Theorem` *nombre* `: prop` `Goal prop`

```
Coq < Lemma ejemplo : forall (A B : Prop), A → B.
1 subgoal
______________________________________(1/1)
forall A : Prop, A → B
```

Como se puede observar, Coq muestra la meta actual en la parte inferior de la linea. En la parte superior se muestra el contexto actual de la demostración, que en este caso es vacío. El contexto es una lista de hipótesis que se asumen como verdaderas. Estas hipótesis pueden ser utilizadas para demostrar la meta.

Para empezar la demostración se utiliza el comando `Proof.`.

```
Coq < Proof.
```

Habiendo iniciado la demostración, se pueden utilizar las tácticas para manipular la meta y el contexto. Cada táctica manipula el contexto actual de la demostración. El contexto es esencial en la demostración en Coq, ya que permite la reutilización de hipótesis y la construcción de pruebas más complejas a partir de hipótesis más simples.

Por ejemplo, después de usar la táctica `intros`, que se verá mas adelante en este capítulo, el contexto se ve de la siguiente manera:

```
1 subgoal
A : Prop
______________________________________(1/1)
B
```

En donde se puede observar que una hipótesis, que indica que A es de tipo `Prop`, ha sido agregada al contexto.

Una vez que no quedan metas por demostrar, se puede terminar la demostración con el comando `Qed.`.

```
Coq < Qed.
ejemplo is defined
```

Así, una demostración usualmente se ve de la siguiente manera.

```
(Theorem|Lemma) ejemplo: prop
Proof.
tactica_1.
tactica_2.
...
tactica_k.
Qed.
```

Existen diversas tácticas[3]que podemos usar en Coq, a continuación se presentan algunas de las más básicas:

- `intros`: Si hay variables cuantificadas universalmente en la meta, al usar esta táctica, éstas se agregan al contexto. También se puede usar para agregar todas las proposiciones en el lado izquierdo de una implicación como hipótesis.

```
------------------------------------(1/1)
forall p q : Prop, (p → q) → ¬q → ¬p

Coq < intros.
p, q : Prop
H : p → q
H0 : ¬q
------------------------------------(1/1)
¬p
```

[3]Se puede consultar la lista completa de tácticas en https://coq.inria.fr/refman/proof-engine/tactics.html.

- `assumption`: Si lo que se está tratando de probar ya está en el contexto, se puede usar esta táctica para terminar la prueba.

```
p : Prop
H : p
--------------------(1/1)
p

Coq < assumption.
No more subgoals.
```

- `simpl`: Esta táctica trata de simplificar términos complejos.

```
--------------------(1/1)
2021 + 2 = 2023

Coq < simpl.
--------------------(1/1)
2023 = 2023
```

- `reflexivity`: Úsese cuando se quiera probar una meta que es una igualdad. Es decir, el objetivo tiene la forma $x = y$ y tanto x como y son iguales sintácticamente después de algunos cálculos.

```
--------------------(1/1)
40 + 2 = 42

Coq < reflexivity.
No more subgoals.
```

- `apply` H: Si la meta es alguna proposición B y se tiene una hipótesis H de la forma $A \to B$, entonces para probar que B se cumple, basta con demostrar que A se cumple. `apply` usa este razonamiento para transformar la meta B en A.

```
p, q : Prop
H : p → q
H0 : p
--------------------(1/1)
q

Coq < apply H.
p, q : Prop
H : p → q
H0 : p
--------------------(1/1)
p
```

- `rewrite` H: Si la meta es una igualdad $x = y$ y se tiene una hipótesis H de la forma $x = z$, entonces se puede usar `rewrite` para reemplazar x por z en la meta.

```
x : nat
H : x = 42
--------------------(1/1)
x + 2 = 44

Coq < rewrite H.
x : nat
H : x = 42
--------------------(1/1)
42 + 2 = 44
```

- `left/right`: Si la meta es una disyunción $A \vee B$, al usar left se reemplaza la meta con el lado izquierdo de la disyunción. De manera análogca ocurre con `right`.

```
p,q : Prop
--------------------(1/1)
p ∨ q

Coq < left
p,q : Prop
--------------------(1/1)
p
```

- `replace` t_1 `with` t_2: Permite reemplazar el término t_1 en la meta con el término t_2. Al usar esta táctica se crea una nueva meta que pide demostrar que esos dos términos son iguales.

```
x : nat
--------------------(1/1)
S (x + 1) = S (S x)

Coq < replace (x + 1) with (S x).
x : nat
--------------------(1/2)
S (S x) = S (S x)
--------------------(2/2)
S x = x + 1
```

- `split`: Si la meta es una conjunción de la forma $A \wedge B$, la táctica reemplaza el objetivo con dos submetas: A y B.

```
p, q : Prop
--------------------(1/1)
p ∧ q

Coq < split.
p, q : Prop
--------------------(1/2)
p
--------------------(2/2)
q
```

- `assert P`: Dado una meta G, `assert P` crea una nueva meta P y agrega P al contexto en el cual se tiene que probar G.

```
p, q : Prop
--------------------(1/1)
p ∧ q

Coq < assert (p → q).
2 subgoals

p, q : Prop
H : p → q
--------------------(1/1)
p ∧ q
--------------------(1/1)
p → q
```

- `unfold` def: Reemplaza un término con su definición.

```
...
--------------------(1/1)
p ↔ ¬q

Coq < unfold iff.
--------------------(1/1)
p → ¬q ∧ ¬q → p.

Coq < unfold not.
--------------------(1/1)
(p → q → False) ∧ ((q → False) → p)
```

- `destruct` H:
  - Si se tiene una hipótesis H de la forma $P \wedge Q$, reemplaza la hipótesis con dos hipótesis: P y Q.

```
H : p ∧ q
--------------------(1/1)
p

Coq < destruct H.
H : p
H0 : q
--------------------(1/1)
p
```

  - Si H es una disyunción $P \vee Q$, genera dos submetas, una en donde se tiene como hipótesis a P y otro donde se tiene como hipótesis a Q.

```
H : p ∨ q
--------------------(1/1)
q ∨ p

Coq < destruct H.
H : p
--------------------(1/2)
q ∨ p
--------------------(2/2)
q ∨ p
...
(* Una vez que se ha probado la primer meta *)
H : q
--------------------(1/1)
q ∨ p
```

- `exfalso`: Implementa el principio lógico *ex falso quodlibet*[4]. Se reemplaza la meta actual con `False`.

[4]También conocido como *principio de explosión*. Establece que a partir de una contradicción se puede demostrar cualquier proposición [9].

```
p, q : Prop
H1 : p → q
H2 : q → False
H3: p
--------------------(1/1)
q

Coq < exfalso.
p, q : Prop
H1 : p → q
H2 : q → False
H3: p
--------------------(1/1)
False
```

- `destruct (classic` P`)`: Se usa el principio del tercero excluido para probar una meta. Se generan dos submetas idénticas, una donde se tiene como hipótesis a P y otra donde se tiene como hipótesis a $\neg P$.

```
p : Prop
H : (p → False) → False
______________________________________(1/1)
p

Coq < destruct (classic p).
2 subgoals
p : Prop
H : (p → False) → False
H0 : p (*Se agrega p como hipótesis *)
______________________________________(1/2)
p
______________________________________(2/2)
p

... (* Una vez que se ha probado la primer meta *)

p : Prop
H : (p → False) → False
H0 : ¬p (*Se agrega ¬p como hipótesis *)
______________________________________(1/1)
p
```

- `exists` x: Si la meta es de la forma $existsy, \varphi$, la táctica $existsx$ permite introducir un nuevo elemento x y reemplaza la meta con $\varphi[y/x]$. Esta táctica se utiliza para demostrar la existencia de un elemento particular que satisface una propiedad.

```
P : nat → Prop
--------------------(1/1)
exists x, P x

Coq < exists 42.
p : nat → Prop
--------------------(1/1)
P 42
```

- `destruct` H: Si H es una hipótesis de la forma $existsx, \varphi$, se reemplaza H con dos hipótesis, una que afirma que x es de un cierto tipo, y otra que afirma que φ se cumple para x.

```
H : exists x : nat, P x
--------------------(1/1)
Q x

Coq < destruct H.
x : nat
H : P x
--------------------(1/1)
Q x
```

A modo de ejemplo, considerese la siguiente proposición:

$$\forall\ n\ m \in \mathbb{N}, n = 0 \land m = 0 \rightarrow n + m = 0$$

Haciendo uso de Coq, podemos generar una demostración para la proposición anterior.

```
Coq < Theorem suma_cero : forall n m : nat, n = 0 ∧ m = 0 → n + m = 0.

--------------------(1/1)
forall n m : nat, n = 0 ∧ m = 0 → n + m = 0
```

Dado que se tienen variables cuantificadas universalmente y, además se tiene una implicación, entonces es posible hacer uso de la táctica `intros`.

```
suma_cero < intros.

n, m : nat
H : n = 0 ∧ m = 0
--------------------(1/1)
n + m = 0
```

Como se tiene una conjunción como hipótesis, se puede usar la táctica `destruct H` para dividirla en dos hipotesis.

```
suma_cero < destruct H.

n, m : nat
H : n = 0
H0 : m = 0
--------------------(1/1)
n + m = 0
```

Tip

Existe una táctica especial que se representa por el símbolo `;`, ésta aplica la táctica del lado derecho del punto y coma a todas las metas producidas por la táctica del lado izquierdo. Así, también es posible encadenar tácticas para hacer el código más conciso.

$$t_1; t_2.$$

Ahora se tiene como hipótesis que tanto n como m tienen un valor de 0, por lo que es posible reemplazar la variable por su valor en la meta usando `rewrite`.

```
suma_cero < rewrite H; rewrite H0.

n, m : nat
H : n = 0
H0 : m = 0
--------------------(1/1)
0 + 0 = 0
```

Usando `simpl` se puede simplificar `0 + 0` a simplemente `0`

```
suma_cero < simpl.

n, m : nat
H : n = 0
H0 : m = 0
--------------------(1/1)
0 = 0
```

Finalmente, usando `reflexivity` se consigue probar que $0 = 0$, terminando así la demostración.

```
suma_cero < reflexivity.
No more subgoals.
```

7.3. Desarrollo de la práctica

Usando las tácticas vistas en este capítulo, da una prueba para las siguientes proposiciones.

Lógica Proposicional

- Contrapositiva: $(p \to q) \to (\neg q \to \neg p)$
- Distributividad de la conjunción: $p \wedge (q \vee r) \leftrightarrow (p \wedge q) \vee (p \wedge r)$
- Eliminación de la doble negación: $\neg\neg p \to p$
- Eliminación de la implicación: $(p \to q) \to (\neg p \vee q)$
- Ley de Morgan para la disyunción: $\neg(p \vee q) \leftrightarrow \neg p \wedge \neg q$
- Ley de Peirce: $((p \to q) \to p) \to p$

Lógica de Predicados

- Distributividad del cuantificador universal sobre la conjunción: $\forall x(P(x) \wedge Q(x)) \leftrightarrow (\forall x P(x) \wedge \forall x Q(x))$
- Distributividad del cuantificador existencial sobre la disyunción: $\exists x(P(x) \vee Q(x)) \leftrightarrow (\exists x P(x) \vee \exists x Q(x))$
- *P es válido para todos los x* si *no hay ningún x para el cual P no sea válido*: $\forall x, Px \to \neg(\exists x, \neg Px)$.
- *Existe un x para el cual P x es válido* si *no es válido que para todo x, P x no sea válido*: $\exists x, Px \to \neg(\forall x, \neg Px)$.

Práctica 8

Inducción en Coq

8.1. Objetivos

- Hacer uso de Coq para definir tipos de datos inductivos, así como funciones sobre éstos.
- Probar propiedades sobre tipos de datos inductivos usando el principio de inducción.

8.2. Introducción

Existen diversos conjuntos y estructuras matemáticas que son definidas de manera recursiva, como lo son los números naturales, las listas, los árboles binarios, entre otras. Para poder probar propiedades sobre éstos se debe hacer uso de su principio de inducción.

8.2.1. Universos

En Coq todo tiene un tipo, incluso los tipos. A los tipos de tipos se les conoce como universos[1], y los más comunes son `Type`, `Set` y `Prop`.

El tipo `Prop` se usa para definir proposiciones y pruebas [2].

[1] En inglés se conocen como *sorts*.

```
Coq> Check forall (n : nat) , n >= 0.
forall n : nat, n >= 0 : Prop
```

El tipo `Set` contiene los tipos cuyos valores son computables [2]. Esto significa que los valores de estos tipos se pueden construir y manipular en la computadora. Los tipos en Set se pueden pensar como tipos de datos ordinarios, como enteros, cadenas o booleanos.

```
Coq> Check nat.
nat : Set
```

Por último, el tipo `Type` es el más general de los tres, y contiene a todos los demás tipos, es decir, aquellos que no están en `Prop` ni en `Set`. Los tipos en `Type` a menudo se utilizan para definir tipos que dependen de otros tipos, como tipos paramétricos.

```
Coq> Check list.
list : Type → Type
```

8.2.2. Tipos de datos recursivos

De manera similar a HASKELL, en COQ es posible definir tipos de datos inductivos mediante la definición de un conjunto de constructores.

La sintaxis para declarar un tipo de manera inductiva es:

```
Inductive nombre (x_1 : t_1) ... (x_n : t_n) : T := C_1 : T_1 | ... | C_n : T_n
```

Código 8.1: Sintaxis para declarar un tipo de manera inductiva

en donde T indica el universo; C_i son los constructores; y T_i es el tipo de los argumentos de cada constructor.

Con esta sintaxis es posible definir tipos de datos recursivos, por ejemplo, los números

naturales son definidos de la siguiente manera en Coq:

```
Inductive nat : Set :=  O : nat | S : nat → nat
```

Código 8.2: Definición del tipo de dato `nat`

La sintaxis para definir un tipo de datos también permite definitr tipos paramétricos. Los tipos parámetricos son aquellos que dependen de otros tipos. Esto permite definir tipos de datos genéricos, como lo son las listas, los árboles, entre otros.

Para definir un tipo de dato paramétrico se deben especificar los argumentos paramétricos en la declaración del tipo, que son los $(x_i : t_i)$ que se observan en el Código 8.1, en donde x_i es una variable de tipo y t_i es el tipo de la variable.

Si por ejemplo, se desea definir listas[2] homogeneas, la definición es como sigue:

```
Inductive lista (A : Type) : Type :=  nil : lista A
                                    | cons : A → lista A → lista A.
```

Código 8.3: Definición de listas en Coq

Tip

Para ver la definición de una función o un tipo de dato se puede usar el comando `Print`.

```
Coq> Print list.
Inductive list (A : Type) : Type := nil : list A
                                  | cons : A → list A → list
```

8.2.3. Coincidencia de patrones

Cuando un término t pertenece a algún tipo recursivo, es posible construir un nuevo término haciendo un análisis de casos sobre los constructores del tipo. Esto es conocido en la

[2]Si se desea profundizar en el manejo de listas, puede consultar la sección *Working with structured data* de la referencia [27]

programación funcional como coincidencia de patrones[3] [26]. La sintaxis es la siguiente:

$$\texttt{match}\ t\ \texttt{with} \mid c_1\ args_1 \Rightarrow term_1 \mid \ldots \mid c_k\ args_k \Rightarrow term_k\ \text{end}$$

en donde t es el término a evaluar y $term_i$ es el término que se produce cuando la evaluación de t hace *match* con el constructor c_i.

Por ejemplo, la función que determina si un número natural es el 0 o no, se define como en el Código 8.4.

```
Definition iszero n := match n with | 0 ⇒ true | S _ ⇒ false end.
```

Código 8.4: Definición de la función `iszero`

Nótese la similitud entre esta sintaxis y la de la expresión `case` de `Haskell`.

8.2.4. Funciones recursivas

Muchas de las funciones que se definen sobre tipos de datos recursivos, son funciones recursivas. En Coq las funciones recursivas no se pueden definir usando `Definition`. En cambio Coq provee la siguiente sintaxis:

```
Fixpoint nombre (x1 : t1) ... (xn : tn) : T := cuerpo
```

Código 8.5: Sintaxis para definir funciones recursivas

en donde los x_i son los argumentos de la función junto a su tipo t_i y T es el tipo que devuelve la función.

Con esta sintaxis y usando *coincidencia de patrones*, es posible definir funciones recursivas sobre cualquier tipo de dato.

A modo de ejemplo, considere la función para sumar dos números naturales y la función para devolver la suma de todos los elementos de una lista de números, mostradas en el Código 8.6 y 8.7 respectivamente.

[3] *pattern matching* en inglés.

```
Fixpoint suma (n m : nat) : nat := match m with | O ⇒ n
                                                | S p ⇒ S(suma n p)
                                   end.
```

Código 8.6: Definición de la función `suma` sobre números naturales.

```
Fixpoint suma_lista (l : list nat) : nat := match l with
                                    | nil ⇒ 0
                                    | cons a m ⇒ a + suma_lista m end.
```

Código 8.7: Definición de la función que calcula la suma de los elementos de una lista.

8.2.5. La táctica `induction`

Muchas veces, para probar propiedades sobre datos recursivos o funciones recursivas se necesita hacer uso de inducción. Para ello existe la táctica `induction`.

Tómese como ejemplo la función `length` que calcula la longitud de una lista, cuya definición puede verse en el Código 8.8.

```
Fixpoint length (l : list A) : nat := match l with
                                      | nil ⇒ 0
                                      | (cons _ l') ⇒ S (length l')
end.
```

Código 8.8: Definición de la función `length`.

Nota

La función `length` ya se encuentra definida en la biblioteca estandard de Coq, por lo que no es necesario definirla nuevamente.

Considére la siguiente proposición:

```
lenght(xs ++ ys) = lenght(xs) + length(ys)
```

que establece que la longitud de la concatenación de dos listas es igual a la suma de la

longitud de cada lista.

Para probar que lo anterior se cumple con ayuda de Coq se procede a hacer una prueba por inducción.

```
Coq < Theorem suma_longitud (A : Type) : forall xs ys : list A,
      length (xs ++ ys) = length xs + length ys.

...
xs, ys : list A
===============(1/1)
length (xs ++ ys) = length xs + length ys
```

La táctica `induction` recibe como argumento una variable sobre la cual se hará inducción, para este ejemplo, se usará `xs`.

```
suma_longitud < induction xs.

ys : list A
===============(1/2)
length (nil ++ ys) = length nil + length ys
===============(2/2)
length ((a :: xs) ++ ys) = length (a :: xs) + length ys
```

Al usar esta táctica se producen tantas submetas como constructores existen en el tipo de dato sobre el cual se esta haciendo inducción, agregando una hipótesis de inducción en el caso de constructores recursivos, tal y como sucede cuando se usa el principio de inducción.

Para probar la primera meta, se puede usar la táctica `simpl.` seguida de `reflexivity`, terminando así la primera parte de la prueba.

```
suma_longitud < simpl; reflexivity.
This subproof is complete, but there are some unfocused goals.
```

Para la segunda meta, se tiene el siguiente contexto:

```
a : A
xs, ys : list A
IHxs : length (xs ++ ys) = length xs + length ys
```

Usando `simpl`, podemos usar la definición de `length`.

```
suma_longitud < simpl.

a : A
xs, ys : list A
IHxs : length (xs ++ ys) = length xs + length ys
===============(1/1)
S (length (xs ++ ys)) = S (length xs + length ys)
```

Dado que en el contexto se tiene que `length (xs ++ ys) = length xs + length ys` entonces se puede rescribir la parte de `length (xs ++ ys)` en la meta.

```
suma_longitud < rewrite IHxs.

a : A
xs, ys : list A
IHxs : length (xs ++ ys) = length xs + length ys
===============(1/1)
S (length xs + length ys) = S (length xs + length ys)
```

Y finalmente, usando `reflexivity` la prueba termina.

```
suma_longitud < reflexivity.
No more subgoals.
```

8.3. Desarrollo de la práctica

Usando las tácticas vistas en este capítulo y en el anterior, da una prueba para las siguientes proposiciones.

Listas

1. Demuestra que la concatenación de listas es asociativa, es decir, que para cualesquiera 3 listas xs, ys y zs se cumple que `xs ++ (ys ++ zs) = (xs ++ ys) ++ zs`

2. Define de manera recursiva la función `reversa` sobre listas. Demuestra las siguientes proposiciones:

   a) `length (reversa xs) = length xs`

   b) `reversa (xs ++ ys) = reversa xs ++ reversa ys`

3. Considere las funciones `take` y `drop` cuya especificación es la siguiente:

   - `take n xs`: Devuelve la lista con los primeros n elementos de xs.
   - `drop n xs`: Devuelve la lista resultante de eliminar los primeros n elementos de xs.

   a) Define de manera recursiva las funciones `take` y `drop`.

   b) Demuestra que `take n xs ++ drop n xs = xs`

Árboles Binarios

1. Define de manera inductiva el tipo de dato `arbol` que corresponde a la estructura de datos conocida como *árbol binario* en la variante donde los elementos de éste se encuentran en los nodos, cuya definición es la siguiente:

   - El árbol *vacío* es un árbol binario.
   - Si $t1$ y $t2$ son árboles binarios y v es un elemento de un tipo A, entonces `Nodo (t1 v t2)` es un árbol binario.

2. Define de manera recursiva la función `refleccion` sobre árboles binarios que intercambia los árboles izquierdo y derecho de cada nodo.

3. Define de manera recursiva la función `altura` que calcula la altura de un árbol binario. La altura de un árbol binario se define como la longitud del camino más largo desde la raíz hasta una hoja.

4. Demuestra que la función `refleccion` es involutiva, es decir, que dado un árbol t, se cumple que `refleccion (refleccion t) = t`.

5. Demuestra que se cumple que `refleccion` preserva la altura de un árbol.

Parte IV

Proyectos

Algoritmo DPLL

9.4. Objetivos

- ▷ Dar una breve introducción al problema de satisfacibilidad booleana.
- ▷ Implementar el algoritmo DPLL *(Davis-Putnam-Logemann-Loveland)* para resolver el problema de satisfacibilidad booleana.

9.5. Introducción

El *Teorema de Cook*[4] introduce el concepto de NP-completud, que establece lo siguiente:

Definición 8 (NP-completud) *Un problema de decisión es NP-completo si pertenece a la clase de complejidad NP y si para todo problema en NP, éste puede ser reducido a él utilizando una transformación polinómica. [33]*

El Teorema de Cook es de gran importancia, pues además de introducir el concepto de *NP*-completud, establece que el problema de satisfacibilidad booleana es un problema *NP*-completo y de hecho, el primero en ser demostrado que pertenece a esta clase de complejidad.

Al ser un problema *NP*-completo el problema de satisfacibilidad booleana, no se sabe si existe un algoritmo eficiente que dada una instancia de este problema, pueda resolverlo. Sin embargo a lo largo de los años se han desarrollado sistemas capaces de "resolver" instancias del problema SAT de manera eficiente. A estos sistemas se les llama *solucionadores SAT*.

Muchos de los *solucionadores SAT* modernos usan como base al algoritmo DPLL [25], que

[4]También conocido como *Teorema de Cook-Levin* pues Leonid Levin lo demostró casi al mismo tiempo que Stephen Cook.

es un algoritmo basado en la técnica de *backtracking*. Con el paso del tiempo, el algoritmo ha sido modificado con el fin de hacerlo más eficiente. Sin embargo, para propósitos didácticos, en este capítulo se revisa una versión simplificada de éste.

9.6. Problema SAT

En su forma más general, el problema de satisfacibilidad booleana, también nombrado *SAT*, busca responder lo siguiente:

> *Dada una fórmula proposicional φ, ¿existe un estado de las variables $\mathcal{I}$ de tal manera que suceda que $\mathcal{I}(\varphi) = 1$?*

A simple vista, podría parecer que el problema SAT es relevante solo para el área de la teoría de la complejidad y para las Ciencias de la Computación en general, no obstante, las aplicaciones en problemas de interés práctico son diversas, problemas tan simples como un sudoku o tan complicados como verificación de hardware, de planeación (control de tráfico aereo, logística, etc.) así como criptoanálisis, pueden ser resueltos mediante un *solucionador SAT*.

Como se recordará, uno de los motivos por los cuales se estudia a la lógica, es poder modelar problemas mediante un lenguaje, de manera que sea posible razonar formalmente sobre estos. A modo de ejemplo, considerese el *problema del palomar*, que establece que es imposible colocar a n palomas en m palomares de tal manera que cada paloma este en un palomar diferente cuando $n > m$. Para los humanos, resolver este problema es trivial, sin embargo, para un solucionador SAT puede llegar a ser un problema muy complejo. Para codificar el problema anterior en una fórmula proposicional primero se debe definir que a las variables proposicionales y su significado. En el caso del problema del palomar, sea $p_{i,j}$ la variable proposicional que representa que la i-ésima paloma se encuentra en el j-ésimo palomar. Ahora, lo que se quiere, es poder codificar que:

- Cada paloma este en un palomar

$$\bigwedge_{1 \leq i \leq n} \bigvee_{1 \leq j \leq m} p_{i,j} \tag{9.1}$$

Con esta fórmula se esta especificando que para cada paloma i, ésta tiene que estar en al menos un palomar.

- No puede haber dos palomas en el mismo palomar

$$\bigwedge_{1\leq j\leq m}\bigwedge_{1\leq i<k\leq n}\neg(p_{i,j}\wedge p_{k,j}) \tag{9.2}$$

Aquí se especifica que para cada palomar j, no puede haber dos palomas en el mismo palomar. Es importante observar que se itera sobre todas las palomas i y k de tal manera que $i < k$, esto es para evitar que se repitan las comparaciones, pues dado que la conjunción es conmutativa, no es necesario comparar $p_{i,j}$ con $p_{k,j}$ y $p_{k,j}$ con $p_{i,j}$.

De manera de que la codificación del problema del palomar consiste en la conjunción de las dos fórmulas descritas anteriormente. Entonces, para determinar si dada una instancia del problema del palomar ésta tiene solución, lo que se haría es codificar la instancia en una fórmula proposicional y con la ayuda de un solucionador SAT determinar si es satisfacible. Si lo es, entonces la interpretación que haga satisfacible a la fórmula también proporcionará la solución, de lo contrario, no existe solución.

9.7. Desarrollo del proyecto

Actualmente, muchos de los solucionadores SAT basan su funcionamiento en alguna variación del algoritmo Davis-Putnam-Logemann-Loveland (DPLL) [25], el cual que sirve para decidir la satisfacibilidad de una fórmula proposicional en forma normal conjuntiva. Aunque existen diversas variantes del algoritmo, todas se basan en el mismo conjunto de reglas.

9.7.1. Implementación del algoritmo DPLL

El procedimiento DPLL es un algoritmo de decisión, es decir, dada una fórmula proposicional éste determina si es o no satisfacible, pero no da el modelo o los modelos que satisfacen a la fórmula. No obstante existe una versión del algoritmo que además de determinar si la fórmula es o no satisfacible, como parte del resultado devuelve un modelo en caso de que la fórmula sea satisfacible. El algoritmo se modela como un sistema de transiciones, en donde

cada estado $M \models_? F$ representa la búsqueda de un modelo M para F. En HASKELL estos estados se puede representar mediante un par, en donde la primer proyección corresponde a una `Interpretacion`, mientras que la segunda proyección corresponde a una `Fórmula`. Al igual que en el Capítulo 3, se trabaja con una fórmula proposicional en forma normal conjuntiva, la cual se modela mediante una lista de claúsulas, que a su vez son listas de literales.

```
type Interpretacion = [(String, Bool)]
type Estado = (Interpretacion, [Clausula])
```

Por otro lado, las transiciones entre estados, que se modelan como $M \models_? F \triangleright M' \models_? F'$, se representan mediante de la aplicación de reglas. Para el algoritmo DPLL, tenemos las siguientes reglas:

- *Conflicto* (`conflict :: Estado -> Bool`): $M \models_? F, \square \triangleright \times$

  Determina si la búsqueda del modelo ha fallado. Dado que no existe modelo que satisfaga a la claúsula vacía, si ésta es parte del conjunto de claúsulas entonces la búsqueda falla.

```
ghci> conflict ([], [[p, q], [r, s], [], [p, r,t]])
True
```

- *Éxito* (`success :: Estado -> Bool`): $M \models_? \varnothing \triangleright \checkmark$.

  Determina si la búsqueda del modelo ha sido exitosa, esto sucede cuando el conjunto de cláusulas es vacío.

```
ghci> success ([(p, True), (q, False)], [])
True
```

- *Claúsula unitaria* (`unit :: Estado -> Estado`): $M \models_? F, \ell \triangleright M, \ell \models_? F$

  Si ℓ es una claúsula unitaria, entonces basta con agregar ℓ al modelo y seguir

con la búsqueda. Es importante notar que si $\ell^c \in M$ esta regla no puede aplicarse, de lo contrario se necesitaria que tanto ℓ como ℓ^c sean verdaderas, lo que lleva a una contradicción.

```
ghci> unit 1 ([], [[p, q], [r, s], [1]])
([(1, True)], [[p,q], [r,s]])
```

- *Eliminación* (`elim :: Estado -> Estado`): $M, \ell \models_? F, \ell \vee C \ \triangleright \ M, \ell \models_? F$

  Si ℓ es una literal que pertenece al modelo M y se tiene la claúsula $\ell \vee C$ entonces, dado que ℓ es verdadera, $\ell \vee C$ también lo es, por lo que se elimina la claúsula $\ell \vee C$ del conjunto de cláusulas.

```
ghci> elim ([(p, True)], [[p, q], [r, s], [p, r, t]])
([(p, True)], [[r,s]])
True
```

- *Reducción* (`red :: Estado -> Estado`): $M, \ell \models_? F, \ell^c \vee C \ \triangleright \ M, \ell \models_? F, C$

  Si ℓ es una literal que pertenece al modelo M y se tiene la claúsula $\ell^c \vee C$ entonces, dado que ℓ es verdadera, ℓ^c es falsa, por lo que solo es de interés saber si C es satisfacible.

```
ghci> red ([(p, True)], [[not p, q], [r, s], [p, r, t]])
([(p, True)], [[q], [r, s], [p, r, t]])
True
```

- *Separación* (`sep :: Literal -> Estado -> (Estado, Estado)`): $M \models_? F \triangleright M, \ell \models_? F; M, \ell^c \models_? F$

  Dada una literal ℓ que figure en alguna claúsula del conjunto de cláusulas, se procede a buscar que $M, \ell \models_? F$, o que $M, \ell^c \models_? F$.
  Observese como en esta regla se generan dos caminos a explorar, uno con M, ℓ y otro con M, ℓ^c. De modo que si el primer camino falla en la búsqueda

se procede a la búsqueda del segundo camino, si éste también falla entonces no hay modelos.

```
ghci> sep p ([], [[p, q]])
(([(p, True)], [[p,q]]), ([(p, False)], [[p,q]]))
True
```

Es importante destacar que el orden en como se aplican las reglas no tiene repercusión en el resultado final, por ejemplo, una estrategía muy usada es aplicar `unit` tantas veces como sea posible. Sin embargo el estado final producido por cualquier estrategia será o un estado fallido cuando la fórmula es insatisfacible, o un estado de la forma $M \models_? F$ donde M es un modelo de F [25]. Sin embargo, se propone aplicar las reglas en el orden listado para hacer la implementación más sencilla, es decir, primero se verifica que no haya algún conflicto usando `conflict`, en caso contrario se verifica si el modelo ha sido encontrado usando `success`, si no es así, entonces se procede a tratar de aplicar `unit` y así sucesivamente.

Usando las funciones definidas anteriormente, se puede implementar el algoritmo DPLL mediante la función principal:

```
dpll :: [Clausula] -> Interpretacion
```

La cual recibe una fórmula proposicional en forma clausular y devuelve una interpretación que satisfaga a la fórmula, obtenida mediante la ejecución del algoritmo DPLL iniciando la ejecución con el estado $\varnothing \models_? F$. En caso de que la fórmula no sea satisfacible, la función deberá devolver una lista vacía.

9.7.2. Aplicación del algoritmo DPLL

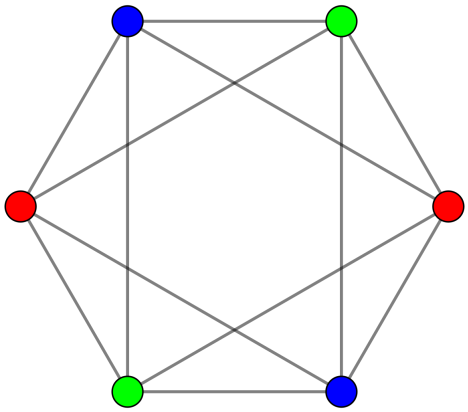
Nota

El material presentado en esta sección se indica como optativo. Esto debido a la complejidad que ya representa la implementación del algoritmo DPLL. Sin embargo, se

recomienda al lector interesado en el tema, que intente resolver los problemas propuestos.

El problema de la k-coloración de gráficas consiste en, dada una gráfica G con n vértices y m aristas, verificar si existe una asignación de k colores de los vértices de la gráfica de manera que para cada dos vertices adyacentes el color asignado sea diferente.

Por ejemplo, la siguiente gráfica esta 3-coloreada.



1. Dé una codificación apropiada para el problema de la k-coloración para una gráfica con v vértices y e aristas.

2. Considerese la siguiente gráfica conocida como la *Gráfica de Petersen*:

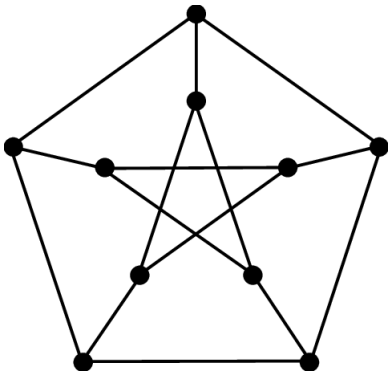


Figura 9.1: Gráfica de Petersen

Usando el algoritmo DPLL implementado en la sección 9.7, determinar lo siguiente:

a) Si existe una 2-coloración de la gráfica.

b) Si existe una 3-coloración de la gráfica.

Punto extra: Cree un codificador SAT para el problema de la k-coloración de gráficas, que dado un archivo que contiene la descripción de una gráfica en formato *DIMACS* regrese la codificación de la gráfica como una fórmula proposicional.

- Investigue sobre el formato *DIMACS* y explique brevemente en que consiste.
- El codificador SAT puede ser escrito en cualquier lenguaje de programación, sin embargo, la fórmula resultante debe ser sintácticamente compatible con Haskell, esto es, la fórmula obtenida debe poder ser pasada como argumento al algoritmo DPLL programado en la sección 9.7 sin necesidad de hacer alguna modificación.
- Codifique la gráfica del punto 2 en el formato DIMACS, luego utilice el codificador SAT para obtener la fórmula proposicional correspondiente y ejecute el algoritmo DPLL para determinar si existe una 3-coloración. Verifique que la respuesta obtenida en este punto es isomorfa a la respuesta obtenida en el punto 2b.

Generador de horarios escolares

10.8. Objetivos

- Resolver un problema del mundo real haciendo uso de la Programación Lógica.
- Poner en práctica los conocimientos obtenidos sobre PROLOG.

10.9. Introducción

El proceso de construcción de horarios es una tarea común y repetitiva para las escuelas de todo el mundo. Éste es un problema que pertence al campo de la optimización combinatoria, que se puede definir como la asignación de un conjunto de materias en un número limitado de intervalos de tiempo (bloques) y salones de clase, sujeto a un conjunto de restricciones, como lo son la disponibilidad de los profesores y los salones de clase, la cantidad de veces que se imparte una materia, entre otras. El principal desafío de este problema radica en el tamaño del espacio de búsqueda de soluciones. Por lo tanto se deben tener en cuenta una gran de variables y restricciones durante el proceso de búsqueda de una solución[5].

10.10. Desarrollo del proyecto

Para el desarrollo de este proyecto, se busca construir un generador de horarios escolares sujeto a algunas restricciones básicas.

La base de conocimientos que servirá para generar un horario contiene los siguientes predicados:

[5]Si se desea obtener más información acerca de este problema, se recomienda consultar [8]

```
clase(C, P, X) // La clase C que es dada por el profesor P debe darse
   X veces a la semana.
dia_libre(P, D) // El profesor P no puede dar clase el día D.
tiempo_libre(C, L) // Indica que la clase C no puede darse en el
   bloque L.
```

Define el predicado `horario(B, D, H)` que es verdadero, si dada una base de conocimientos con los predicados anteriormente descritos, se cumple que `H` es un horario que contiene el máximo número de clases, de ser posible todas las disponibles, que se distribuyen en a lo más `D` días con un máximo de `B` bloques por día, cumpliendo las restricciones indicadas por los predicados `clase`, `dia_libre` y `tiempo_libre`.

Por ejemplo, la consulta `horario(7,5,H)` busca un horario que a lo largo de 5 días, distribuya las materias disponibles en 7 bloques diarios. Así, con la siguiente base de conocimientos, el resultado de la consulta debería instanciar en `H` un horario similar al mostrado en la Figura 10.2.

```
clase(algebra, p1, 3).
clase(cc, p2, 5).
clase(edd, p2, 4).
clase(proba, p3, 2).
clase(logica, p4, 4).
clase(arqui, p5, 3).
clase(mate, p6, 2).
clase(cripto, p7, 3).
clase(graficas, p8, 4).
dia_libre(p1, lunes).
tiempo_libre(algebra, 1).
```

Lunes	Martes	Miércoles	Jueves	Viernes
cc	cc	cc	cc	cc
edd	algebra	algebra	algebra	mate
graficas	edd	edd	edd	
logica	graficas	graficas	graficas	
arqui	logica	logica	logica	
cripto	arqui	proba	proba	
mate	cripto	arqui	cripto	

Figura 10.2: Horario que representa una respuesta de la consulta `horario(7,5,H)`

Nota

La representación del horario se deja al criterio del alumno. Sin embargo se propone hacer uso de una matriz, es decir, una lista de listas.

De igual manera, se recomienda representar los días de la semana como números para facilitar la manipulación de los datos.

```
?- horario(7,5,H).
H = [
    [cc, edd, graficas, logica, arqui, cripto, mate],
    [cc, algebra, edd, graficas, logica, arqui, cripto],
    [cc, algebra, edd, graficas, logica, proba, arqui],
    [cc, algebra, edd, graficas, logica, proba, cripto],
    [cc, mate, _, _, _, _, _]
].
```

Verificación formal del sistema binario de números naturales[6]

11.11. Objetivos

- Definir y demostrar ciertas propiedades de equivalencia sobre sistemas de representación de números naturales.
- Poner en práctica los conocimientos obtenidos sobre el asistente de pruebas Coq.

11.12. Introducción

Es indiscutible hoy la influencia que tiene en la industria y en casi todos los ámbitos de la vida cotidiana el uso del software. Con el paso del tiempo éste se ha vuelto cada vez mas complejo, haciendo que la introducción de errores sea más común y su detección aún más complicada. En las aplicaciones críticas, que van desde aquellas que tratan con vidas humanas hasta aquellas que manejan grandes cantidades de dinero, la certeza de que el sistema es correcto se considera un criterio indispensable.

En la actualidad, el método más usado para validar software es el *testing*, que se basa en la ejecución de pruebas para verificar el comportamiento del software en diferentes situaciones y compararlo con los requisitos especificados. No obstante, este método no garantiza la corrección del software analizado, por ser incompleto en la mayoría de los casos [11].

Por otra parte se encuentran los métodos formales, que son un enfoque sistemático para el diseño y desarrollo de sistemas que se basa en principios matemáticos y lógicos. Estos

[6]Propuesto originalmente en el capítulo *Induction* de [27].

métodos utilizan técnicas como especificación formal, modelado formal, prueba de software y razonamiento formal para garantizar que un sistema cumpla con ciertos requisitos y propiedades deseadas [23]. Los métodos formales se basan en la idea de que los sistemas pueden ser expresados y analizados de manera precisa y rigurosa mediante lenguajes matemáticos y lógicos. Entre estos métodos se encuentra la *verificación formal*, que es una técnica que se centra en demostrar matemáticamente la corrección de un sistema en relación con ciertas propiedades específicas [32].

11.13. Desarrollo del proyecto

Considere una representación alternativa para los números naturales, la cual se basa en un sistema binario. Es decir, en vez de tener un solo constructor, además del cero, para construir números naturales, se tienen 2 constructores, y cuyas reglas de construcción son las siguientes:

1. El *cero* es un número natural.
2. Si n es un número natural, el *doble* de n también es un número natural.
3. Si n es un número natural, entonces uno mas que n es un número natural.

Dado que se esta proponiendo una definición alternativa a la representación clásica de los números naturales, debería de existir una equivalencia entre ambas representaciones. El propósito de este proyecto es verificar que ambos sistemas son equivalentes mediante la prueba de algunas propiedades haciendo uso de Coq.

Nota

Para el desarollo de este proyecto solo se pueden usar las tácticas vistas a lo largo de este trabajo. Adicionalmente, es posible usar la táctica `omega`, siempre y cuando el uso de ésta no solucione algún ejercicio directamente. El uso de esta táctica deberá estar justificado.

1. Definir de manera recursiva el tipo de dato `bin` que corresponde a la representación de los números naturales en sistema binario.

```
Inductive bin : Type := ...
```

2. Definir la función `incrementa` que suma 1 a un número en forma binaria.
3. Definir la función `bin2nat` que dado un número en forma binaria devuelve su representación en natural.
4. Demuestra que las funciones `incrementa` y `bin2nat` pueden conmutar, es decir, incrementar un número binario y luego convertirlo a natural produce el mismo resultado que convertirlo primero a natural y luego incrementarlo.

```
Theorem conmuta : forall (b: bin),
    bin2nat (incrementa b) = (bin2nat b) + 1.
```

5. Definir la función `nat2bin` que dado un número en forma natural devuelve su representación en binario.
6. Demuestra que al convertir cualquier número natural a binario y luego volver a convertirlo da como resultado el mismo número natural con el que se comenzó.

```
Theorem inversa : forall (n: nat),
    bin2nat (nat2bin n) = n.
```

7. Demuestra o da un contraejemplo de la siguiente proposición.

```
Theorem inversa_2 : forall (b: bin),
     nat2bin (bin2nat n) = n.
```

8. Definir la función `doble` que dado un número binario devuelve el doble de éste usando la misma representación.
9. Demuestra las siguientes propiedades:

a) `incrementa (incrementa (doble b)) = doble (incrementa b)`

b) `nat2bin (n + n) = doble (nat2bin n)`

c) `nat2bin (n + n + 1) = ?? (nat2bin n)` donde `??` representa el constructor correspondiente a la sentencia *"Si n es un número natural, entonces uno mas que n es un número natural"*

10. El hecho de que convertir un número binario a natural y luego a binario de nuevo no necesariamente dé el mismo número binario, se debe a que la multiple representación del cero en binario. Definir la función `normaliza` tal que para cualquier número binario *b*, la conversión a natural y luego nuevamente a binario dé como resultado `(normalizar b)`. Demuestra que lo anterior se cumple, es decir, que `nat2bin (bin2nat b) = normaliza b`

    Sugerencia: Use la definición de la función `doble`.

```
Fixpoint normaliza (b: bin): bin := ...
...
```

Conclusiones

En este trabajo se presenta una propuesta para complementar la enseñanza de la asignatura de Lógica Computacional. A través de una serie de prácticas y proyectos, los estudiantes tendrán la oportunidad de aplicar los conceptos teóricos y desarrollar habilidades prácticas en la programación funcional y la programación lógica, además de tener un primer acercamiento a los asistentes de pruebas y el campo de la verificación formal.

A lo largo de 8 prácticas se han cubierto los temas clave de la asignatura, tales como la Lógica Proposicional, la Lógica de Primer Orden, la Programación Lógica y el Razonamiento Ecuacional y Deducción Natural. Los 3 proyectos presentados permiten a los estudiantes aplicar los conocimientos adquiridos, además de que son una excelente oportunidad para que los estudiantes demuestren sus habilidades y apliquen sus conocimientos en un contexto más amplio, permitiendo desarrollar una comprensión más profunda de la Lógica Computacional y de cómo puede ser aplicada en situaciones reales.

Si bien, a lo largo del manual se indica que los ejercicios propuestos sean resueltos ya sea con Haskell, Prolog o Coq, según sea el caso, nada impide que dichos ejercicios sean adaptados a otros lenguajes de programación u otro asistente de pruebas.

Aún cuando el contenido de este manual esta dirigido a profesores y estudiantes de imparten o cursan la asignatura de Lógica Computacional, éste puede ser usado en otras asignaturas como material introductorio o de referencia para cursos como Programación Declarativa, Semántica y Verificación y Razonamiento Automatizado.

La combinación de teoría y práctica es esencial para el aprendizaje efectivo y este manual de prácticas brinda precisamente eso a los estudiantes, por lo que se espera que este trabajo sea una herramienta útil en la enseñanza de la asignatura de Lógica Computacional, ya que este proporciona una serie de ejercicios para que los estudiantes de la asignatura los realicen y pongan en práctica los conceptos adquiridos en las clases teóricas impartidas por el profesor titular del curso, permitiendo así alcanzar los objetivos establecidos en este manual y en la

materia.

Bibliografía

[1] Yves Bertot. «Coq in a Hurry». En: *CoRR* abs/cs/0603118 (2006). arXiv: cs/0603118. URL: http://arxiv.org/abs/cs/0603118.

[2] Yves Bertot y Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. 1st. Springer Publishing Company, Incorporated, 2010. ISBN: 3642058809.

[3] Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2014. DOI: 10.1017/CBO9781316092415.

[4] William F. Clocksin y Christopher S. Mellish. *Programming in Prolog*. 5.ª ed. Berlin: Springer, 2003. ISBN: 978-3-540-00678-7. DOI: 10.1007/978-3-642-55481-0.

[5] Stephen A. Cook. «The Complexity of Theorem-Proving Procedures». En: STOC '71. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, págs. 151-158. ISBN: 9781450374644. DOI: 10.1145/800157.805047. URL: https://doi.org/10.1145/800157.805047.

[6] J. C. Corbin y Michel Bidoit. «A Rehabilitation of Robinson's Unification Algorithm». En: *IFIP Congress*. 1983.

[7] H. Conrad Cunningham. *Notes on Functional Programming with Haskell*. https://john.cs.olemiss.edu/~hcc/csci555/notes/haskell_notes.pdf. 2014.

[8] Abdelkarim Elloumi, Hichem Kamoun, Bassem Jarboui y Abdelaziz Dammak. «The classroom assignment problem: Complexity, size reduction and heuristics». En: *Applied Soft Computing* 14, Part C (ene. de 2014), págs. 677-686. DOI: 10.1016/j.asoc.2013.09.003.

[9] Favio E. Miranda, A. Liliana Reyes, Lourdes del C. González y Selene Linares. *Lógica Computacional*. Notas de clase. Facultad de Ciencias, Universidad Nacional Autónoma de México, 2018.

[10] Michael Genesereth y Eric J. Kao. *Introduction to Logic*. 3rd. Morgan y Claypool Publishers, 2016. DOI: 10.2200/S00734ED2V01Y201609CSL008.

[11] Carlo Ghezzi, Mehdi Jazayeri y Dino Mandrioli. *Fundamentals of software engineering (2. ed.)*. Ene. de 2003. ISBN: 978-0-13-305699-0.

[12] Mohammad Haji-Abdolhosseini y Gerald Penn. *TRALE Reference Manual*. http://www.ale.cs.toronto.edu/docs/ref/ale_ref.pdf. 2003.

[13] Michael Huth y Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. USA: Cambridge University Press, 2004. ISBN: 052154310X.

[14] Graham Hutton. *Programming in Haskell*. 2nd. USA: Cambridge University Press, 2016. ISBN: 1316626229.

[15] José A. Alonso Jimenez. *Logica en Haskell*. https://www.cs.us.es/~jalonso/publicaciones/2007-Logica_en_Haskell.pdf. 2008.

[16] Robert Kowalski. «Algorithm = Logic + Control». En: *Commun. ACM* 22.7 (1979), págs. 424-436. ISSN: 0001-0782. DOI: 10.1145/359131.359136. URL: https://doi.org/10.1145/359131.359136.

[17] Robert Kowalski. *Computational Logic and Human Thinking: How to Be Artificially Intelligent*. Cambridge University Press, 2011. DOI: 10.1017/CBO9780511984747.

[18] Miran Lipovaca. *Learn You a Haskell for Great Good! A Beginner's Guide*. 1st. USA: No Starch Press, 2011. ISBN: 1593272839.

[19] Zohar Manna y Richard Waldinger. *The Logical Basis for Computer Programming. Volume 1: Deductive Reasoning*. USA: Addison-Wesley Longman Publishing Co., Inc., 1985. ISBN: 0201182602.

[20] *Manual de SWI-Prolog*. Web. Consultado el 7 de Noviembre de 2022. https://www.swi-prolog.org/download/stable/doc/SWI-Prolog-8.2.1.pdf.

[21] Simon Marlow y col. «Haskell 2010 language report». En: *Available online http://www.haskell.org/(2011)* (2010).

[22] John C. Mitchell. *Concepts in Programming Languages*. Cambridge University Press, 2002. DOI: 10.1017/CBO9780511804175.

[23] NASA. «Langely Formal Methods». En: (). URL: https://shemesh.larc.nasa.gov/fm/fm-what.html.

[24] Anil Nerode y Richard A Shore. *Logic for applications*. Springer, New York, NY, 1997.

[25] Robert Nieuwenhuis, Albert Oliveras y Cesare Tinelli. «Abstract DPLL and Abstract DPLL Modulo Theories». En: *In LPAR'04, LNAI 3452*. Springer, 2005, págs. 36-50.

[26] Christine Paulin-Mohring. «Introduction to the Coq Proof-Assistant for Practical Software Verification». En: (2012). DOI: 10.1007/978-3-642-35746-6_3.

[27] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg y Brent Yorgey. *Logical Foundations*. Software Foundations series, volume 1. Version 6.1. https://softwarefoundations.cis.upenn.edu/lf-current. Electronic textbook, 2021.

[28] *Plan de estudios del curso de Lógica Computacional*. Web. Consultado el 6 de abril de 2021. http://www.fciencias.unam.mx/asignaturas/1427.pdf.

[29] Eric S. Raymond. *The New Hacker's Dictionary*. 3rd. MIT Press, 1996. ISBN: 978-0-262-68092-9. URL: https://books.google.com/books?id=g80P_4v4QbIC&pg=PA432.

[30] Steve Reeves y Michael Clarke. *Logic for computer science*. Ene. de 1990. ISBN: 978-0-201-41643-5.

[31] Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. 5th. McGraw-Hill Higher Education, 2002. ISBN: 0072424346.

[32] Edgar Serna M. y Morales-V David. «State of the Art in the Research of Formal Verification». En: *Ingeniería, Investigación y Tecnología* 15 (2014), págs. 615-623. DOI: 10.1016/S1405-7743(14)70659-6.

[33] Michael Sipser. *Introduction to the Theory of Computation*. 1st. International Thomson Publishing, 1996. ISBN: 053494728X.

[34] Manuel Soto Romero. «Manual de Prácticas para la Asignatura de Programación Declarativa». Tesis doct. Sep. de 2019. DOI: 10.13140/RG.2.2.12537.42084.

[35] G.J. Summers. *Logical Deduction Puzzles*. Mensa® Series. Sterling, 2006. ISBN: 9781402721335. URL: https://books.google.com.mx/books?id=93dkEd4k6h0C.

[36] G. S. Tseitin. «On the Complexity of Derivation in Propositional Calculus». En: *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*. Ed. por Jörg H. Siekmann y Graham Wrightson. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, págs. 466-483. ISBN: 978-3-642-81955-1. DOI: 10.1007/978-3-642-81955-1_28. URL: https://doi.org/10.1007/978-3-642-81955-1_28.

[37] Canek Pelaez Valdez. *Estructuras de Datos con Java moderno*. Ed. por Prensas de Ciencias. 2018. ISBN: 9786073009666. URL: http://132.248.161.133:8080/jspui/handle/123456789/5912.