DEPARTAMENTO DE SISTEMAS Y COMPUTACIÓN

REPORTE FINAL DEL PERÍODO SABÁTICO

13 de Agosto de 2018 al 12 de Agosto de 2019

Programa Académico No. 5

Elaboración de Material Didáctico para la Enseñanza

Título del Proyecto

Elaboración de Apuntes: Manual de Prácticas: “Programación Lógica y Funcional”

Presenta

César Enrique Rose Gómez

Dictamen No. AS-2-109/2018

Hermosillo, Sonora; 12 de Agosto de 2019

Agradecimientos

Al Tecnológico Nacional de México y al Instituto Tecnológico de Hermosillo por permitir que los docentes e investigadores, tengamos la oportunidad de aportar al proceso de enseñanza-aprendizaje con los productos desarrollados en el período sabático, con el objetivo de colaborar en una mejora continua de las funciones sustantivas de la institución.

Tabla de contenido

Manual de Prácticas de Programación Lógica y Funcional

Objetivo General del Manual

Aplicar los principios lógicos y funcionales de la programación para la resolución de problemas, a través del diseño de programas bajo el paradigma de programación funcional y el paradigma de la programación lógica.

Justificación

La Academia Mexicana de la Computación, establece que "actualmente se vive un creciente interés en el área de Inteligencia Artificial y, en particular, en los sistemas de aprendizaje e inteligencia computacional.
Esto se debe en parte a los avances recientes del área, impulsados por la gran cantidad de datos generados y disponibles para su análisis, los avances en las computadoras para procesar cada vez más rápidamente información y en las aplicaciones comerciales que se están desarrollando utilizando algoritmos maduros de aprendizaje e inteligencia artificial. En general se espera un futuro con inteligencia incremental en muchas áreas, como ciudades inteligentes, carreteras inteligentes, edificios inteligentes, vehículos autónomos extremadamente inteligentes, sistemas administrativos, educativos, legales y gubernamentales inteligentes, así como variantes de teléfonos, televisores y computadores inteligentes".

Utilidad del Manual

Para enfrenar los retos previamente mencionados, se requiere de preparar a los estudiantes con los fundamentos y herramientas que les permitan la solución de problemas complejos. Se considera que el manual tiene una doble utilidad, la primera de ellas, permite tener una definición estándar de las prácticas, bien estructuradas, para que se comprendan adecuadamente los temas y se puedan alcanzar las competencias de la asignatura.
Asimismo, se considera que es un producto que permanece por siempre, colaborando en la transferencia del conocimiento hacia nuevas generaciones y también hacia los propios profesores.
Finalmente, es importante señalar que el aprendizaje del paradigma declarativo de programación de la materia de Programación Lógica y Funcional es de vital importancia para la materia de Inteligencia Artificial, esto es, la práctica de la primera asignatura es una habilidad que se usará en la segunda asignatura.

Programación Funcional

Apuntes y Prácticas

Práctica 1: Paradigma de la programación declarativa

1.1 Competencias a desarrollar

Identificar los paradigmas y lenguajes de la programación representativa.

Capacidad de análisis y síntesis.

Habilidad para buscar y analizar información proveniente de fuentes diversas.

Habilidades de investigación.

Capacidad de aprender.

1.2 Objetivo de la Práctica

Identificar las características de los lenguajes pertenecientes al paradigma de programación declarativa.

1.3 Introducción

Un paradigma de programación se caracteriza por un modelo formal de lo que constituye un cálculo, considerando la definición de cálculo desde dos enfoques; el primero de ellos en un sentido amplio: Cualquier proceso automático de adquisición, manipulación, y difusión de información. El otro enfoque es en un sentido formal: Cualquier proceso automático cuyos aspectos relevantes puedan ser modelados matemáticamente con una máquina de Turing.

El conjunto de cálculos realizables es igual en los siguientes tres casos de paradigmas (los que se pueden hacer con máquinas de Turing) y que constituyen tres categorías de lenguajes de programación:

- Imperativos
- Funcionales
- Lógicos

En la programación imperativa, un cálculo es un conjunto de instrucciones que establecen explícitamente como se debe manipular la información digital presente en memoria, y/o como se debe recoger o enviar información desde/hacia los

dispositivos, ejemplos de estos lenguajes imperativos son: Fortran, C, Java, C++ y C#.

En la programación funcional, un cálculo es el proceso de aplicar una función recursiva a un valor de su dominio para obtener el correspondiente valor del rango (el resultado). El término función recursiva debe entenderse aquí según se introduce en la teoría de la computabilidad, es decir, como una función calculable con una máquina de Turing (no como un subprograma que se invoca a si mismo), ejemplos de lenguajes funcionales son: Lisp, Scheme, ML, Miranda, Haskell y Erlang.

En la programación lógica, un cálculo es el proceso de encontrar que elementos de un dominio cumplen determinada relación definida sobre dicho dominio, o bien determinar si un determinado elemento cumple o no dicha relación. Un programa en estos lenguajes consiste en una especificación de la relación que queremos calcular y normalmente, dicha relación estará especificada en términos de otras, que también se incluyen en el programa. El lenguaje más popular de este tipo de programación es el lenguaje Prolog.

La división principal reside en el enfoque imperativo (indicar el cómo se debe calcular) y el enfoque declarativo (indicar el qué se debe calcular). El enfoque declarativo tiene varias ramas diferenciadas: el paradigma funcional, el paradigma lógico, la programación reactiva y los lenguajes descriptivos. Otros paradigmas se centran en la estructura y organización de los programas, y son compatibles con los fundamentales: Ejemplos de este paradigma es la Programación estructurada, modular, orientada a objetos, orientada a eventos, programación genérica. Por último, existen paradigmas asociados a la concurrencia y a los sistemas de tipado.

En particular la programación declarativa se caracteriza porque los programas describen relaciones entre objetos/valores; pueden ser relaciones funcionales o relaciones más generales, y dejan que el ejecutor del lenguaje (intérprete o compilador) trate de satisfacerlas aplicando un algoritmo fijo de cómputo.

1.4 Correlación con el o los temas y subtemas del programa de estudio vigente

Tema: Conceptos Fundamentales

Subtema: Estilos de programación.

1.5 Material y equipo necesario

Equipo de cómputo e Internet.

1.6 Metodología

Las actividades de aprendizaje que se consideran en esta práctica son las siguientes:

1. Visualizar la evolución de los lenguajes de programación, para ello se recomienda acceder a la siguiente dirección: https://www.levenez.com/lang/.

2. A partir de la identificación de los lenguajes de programación declarativa, investigar cuales son sus principales características y construir un mapa conceptual partiendo del concepto de "Programación declarativa".
3. Lectura del articulo "Why Functional Programming Matters" de John Hughes, el cual puede encontrar en: http://www.cse.chalmers.se/~rjmh/Papers/whyfp.html
4. Instalación de Haskell, como casi todos los otros lenguajes, viene en dos presentaciones: compilada (compilador) e interactiva (intérprete). Un sistema interactivo ofrece una línea de comandos donde se puede experimentar y evaluar expresiones directamente, por lo que es una buena elección para comenzar. Descárguelos de la siguiente dirección: https://www.haskell.org/ghc/.

1.7 Sugerencias didácticas

Después de presentar las características de los diversos paradigmas de la programación, plantear al grupo una reflexión acerca del porque de la existencia de los diversos lenguajes de programación.

1.8 Evidencias de aprendizaje (Reporte del alumno)

Mapa conceptual de las características de los lenguajes declarativos.

Interprete y/o compilador de Haskell instalado.

1.9 Bibliografía

Kenneth C. Louden. (2004). Lenguajes de programación: principios y práctica. Estados Unidos: Thompson Learning.

Sestoft Peter, (2017), Programming Language Concepts, Springer International Publishing.

Kurt Will, (2018), Get Programming with Haskell , Manning Publications.

Programación Funcional

Apuntes y Prácticas

Práctica 2: Introducción y semántica operacional de la programación funcional

2.1 Competencias a desarrollar

Identificar los elementos de la programación funcional.

Habilidad para buscar y analizar información proveniente de fuentes diversas.

Solución de problemas.

Capacidad de aplicar los conocimientos en la práctica.

Habilidades de investigación.

Capacidad de aprender.

2.2 Objetivo de la Práctica

Desarrollar las bases del estilo de programación funcional usando Haskell.

2.3 Introducción

Como previamente se mencionó, existen diversos modelos de cómputo:

1. Máquinas de Turing (formalismo de estilo operacional, máquinas con estado).
2. Teoría de funciones recursivas (formalismo abstracto).
3. Cálculo lambda (λ). Introducido por Church en los años 30, orientado a programas en vez de a la máquina. También es un formalismo abstracto. Inspiró los lenguajes funcionales (Lisp, ML, Haskell, entre otros).

De tal manera que los métodos en Java, las funciones de C, C++ o Pascal, las funciones de Haskell, todos ellos comparten la misma idea de abstracción funcional: bloques de código parametrizados por uno o más argumentos. Después llegaron los módulos, las clases y los objetos, los tipos, etc.

Programas como funciones

Un programa es una descripción de un cómputo especifico. Si se ignoran los detalles del cómputo, el "cómo" del cómputo, y se enfoca en el resultado siendo computado, el "qué" del cómputo, entonces un programa viene a ser simplemente una "caja negra" para obtener una salida de una entrada. Desde este punto de vista, un programa es esencialmente equivalente a una función matemática:

$$y = f(x) \quad o \quad f: X \to Y$$

donde el conjunto X es llamado el dominio de f, mientras el conjunto Y es llamado el rango de f .

En la programación funcional pura se tienen las siguientes características:

- No hay variables.
- Sólo constantes, parámetros y valores.
- No hay estatutos de asignación.
- No hay iteración, solo recursión.
- No hay noción del estado interno de una función: el valor de cualquier función depende sólo de los valores de sus parámetros y no de cualquier cómputo previo, incluyendo llamadas a la propia función.

Otro concepto importante a considerar es el de funciones de alto orden, esto es, una función que usa funciones como argumentos o resultados.

Cálculo Lambda

El cálculo lambda es una teoría que provee reglas para manipular funciones en una manera puramente sintáctica. Ha tenido enorme influencia en los siguientes campos:

• Lógica Matemática.

• Teoría de la Computabilidad.

• Semántica Formal (Denotacional) de los lenguajes de programación.

• Programación Funcional: todos los lenguajes funcionales pueden ser vistos como variaciones sintácticas del cálculo lambda. Además, sus semánticas y sus implementaciones pueden ser analizadas en el contexto del cálculo lambda.

La notación λ de Church permite la definición de funciones anónimas, es decir, funciones sin nombres asociados:

$\lambda n.n^3$ define la función que asocia a cada n en el dominio con n^3.

Diremos que la expresión representada por $\lambda n.n^3$ es el valor ligado con el identificador cubo.

El número y orden de los parámetros de la función son especificados entre el símbolo λ y una expresión. No hay ambigüedad porque se especifica el orden de los parámetros:

$\lambda n.\lambda m.n^2 + m$ que es diferente a $\lambda m.\lambda n.n^2 + m$

La mayoría de los lenguajes funcionales permiten funciones anónimas. Por ejemplo:

$(lambda\ (n)(* n\ n\ n))$	Scheme
$fn\ n\ \Rightarrow n * n * n$	Ml
$function\ n\ \rightarrow n$ ^ 3	Caml
$\backslash\ n\ \rightarrow n$ ^ 3	Haskell

La utilidad del cálculo lambda proviene de poseer una sintaxis y semántica muy sencillas y aún así tener suficiente poder para representar todas las funciones computables.

Las expresiones lambda pueden ser:

1. Variables, que denotaremos utilizando letras minúsculas.
2. Constantes predefinidas, que actúan como valores y operaciones. Estas constantes sólo son permitidas en el cálculo lambda impuro o aplicado.
3. Aplicaciones de funciones.
4. Abstracciones λ (definiciones de funciones).

El cálculo lambda puro no tiene constantes predefinidas pero permite la definición (a través de funciones) de todas las constantes y funciones aritméticas y de manipulación de listas más comunes.

En el cálculo lambda impuro se permiten constantes predefinidas, tales como:

• Numerales. Por ejemplo: 33, 10, etc.

• Funciones predefinidas. Por ejemplo: sum (suma), succ (sucesor), etc.

En las abstracciones: Si V es una variable y E es una λ-expresión entonces λV.E es una abstracción con una variable ligada V y un cuerpo E. De tal manera que la abstracción denota la función que toma como argumento α y regresa como resultado la función denotada por E en un entorno en el cual la variable V denota α.

Una expresión lambda tiene como significado la expresión que resulta después que todas sus aplicaciones funcionales son llevadas a cabo. La evaluación de una expresión lambda se llama reducción.

La regla básica de reducción consiste en sustituir variables libres por expresiones en una manera similar a la forma que los parámetros en una definición de función son pasados como argumentos en una llamada de función.

Notación Currificada en el Lenguaje Haskell

Consideremos la función suma que dados dos enteros devuelve la suma de ambos.

¿Cuál es el tipo de esta función, es decir el conjunto de pertenencia?

Usualmente, la respuesta es:

$$suma: (Z + Z) \rightarrow Z$$

Esto se interpreta como: suma es una función que recibe un par ordenado de números enteros y devuelve un número entero. Por lo tanto suma es una función de un único parámetro o argumento.

Las funciones con múltiples argumentos, pueden manejarse de otra forma, un poco más compleja quizás, para explotar la capacidad de retornar funciones como resultado o de tomar funciones como parámetros.

Se considera la siguiente definición del tipo de suma:

$$suma: Z \rightarrow Z \rightarrow Z$$

Esto se interpreta como: suma es una función que recibe un entero, seguido de otro entero y devuelve la suma de ambos. Esto se deduce de la notación utilizada para dar el tipo de la función $Z \rightarrow Z \rightarrow Z$.

Esta forma de denotar los tipos de las funciones se denomina currificada. El nombre proviene de la persona que popularizó su uso: Haskell Curry. La notación currificada no es simplemente una notación, agrega poder de expresividad a los tipos de las funciones.

El símbolo → (que se lee "implica"), asocia hacia la derecha, esto significa que las dos expresiones siguientes son equivalentes:

$suma: Z \rightarrow Z \rightarrow Z$

$suma: Z \rightarrow (Z \rightarrow Z)$

Por lo tanto se puede decir también que suma es una función que recibe un entero y devuelve una función que a su vez, recibe un entero y devuelve un entero.

Para denotar la aplicación de una función *f* a un parámetro *x*, no usamos la notación usual *f(x)*, sino que lo denotamos como *(f x)*. Dependiendo de la precedencia que se le quiera dar a la aplicación serán necesarios o no los paréntesis.

Para la generalización de la notación currificada a la aplicación de una función con más de un parámetro, se considera lo siguiente: en la sintaxis currificada no se utilizan n-uplas para pasarle parámetros a la función, sino que los mismos "se van pasando de a uno" a la función. A continuación se muestran algunos ejemplos con la sintaxis tradicional y con la sintaxis currificada.

Matemáticas	Haskell
g(x)	*g x*

f(x,y)	*f x y*
g(f(x,y)	*g (f x y)*
f (x, g(y))	*f x (g y)*
g(x+y)	*g (x+y)*
g(x)+y	*g x + y*

Reducción de Expresiones

La labor de un evaluador es calcular el resultado que se obtiene al simplificar una expresión utilizando las definiciones de las funciones involucradas.

Una expresión se reduce sustituyendo, en la parte derecha de la ecuación de la función, los parámetros formales o argumentos por los que aparecen en la llamada (también llamados parámetros reales o parámetros). Cuando una expresión no pueda reducirse más, se dice que está en forma normal.

Es importante el orden en el que se aplican las reducciones, y dos de los mas interesantes son: aplicativo y normal.

En el orden aplicativo se reduce siempre el término más interno (el más anidado en la expresión). En caso de que existan varios términos a reducir (con la misma profundidad) se selecciona el que aparece más a la izquierda de la expresión. Esto también se llama "paso de parámetros por valor", ya que ante una aplicación de una función, se reducen primero los parámetros de la función. A los evaluadores que utilizan este tipo de orden, se les llama "estrictos o impacientes".

El orden normal consiste en seleccionar el término más externo (el menos anidado), y en caso de conflicto el que aparezca más a la izquierda de la expresión. Esta estrategia se conoce como "paso de parámetro por nombre o referencia", ya que se pasan como parámetros de las funciones expresiones en vez de valores. A los evaluadores que utilizan el orden normal se les llama "no estrictos". Una de las características más interesantes es que este orden es normalizante.

Con una estrategia no estricta la reducción de argumentos puede repetirse, por ejemplo la expresión *cuadrado (cuadrado 3)*, la expresión *(3 * 3)* se calcula dos veces como se muestra en la figura 1.

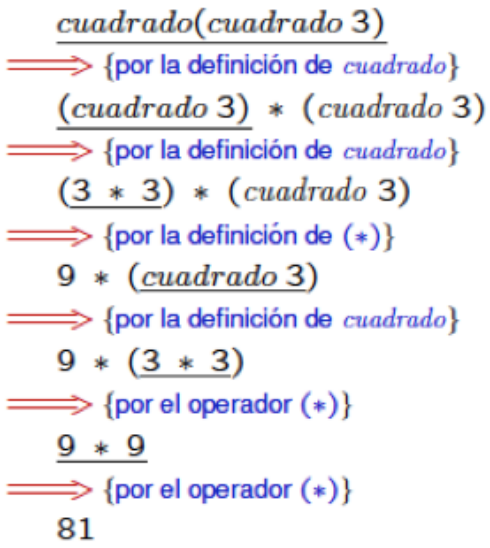


Figura 1. Evaluación no estricta

La evaluación perezosa (lazy) soluciona el problema anterior, no se evalúa ningún elemento en ninguna función hasta que no sea necesario. Las listas se almacenan internamente en un formato no evaluado. La evaluación perezosa consiste en utilizar paso por nombre y recordar los valores de los argumentos ya calculados para evitar recalcularlos. También se denomina estrategia de pasos de parámetros por necesidad. En la figura 2 se muestra el caso anterior bajo la evaluación perezosa.

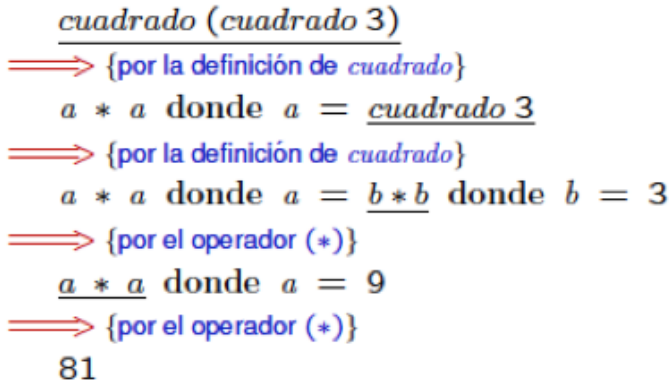


Figura 2. Evaluación perezosa

2.4 Correlación con el o los temas y subtemas del programa de estudio vigente

Tema: Conceptos Fundamentales

Subtemas: Evaluación de expresiones y definición de funciones.

2.5 Material y equipo necesario

Equipo de cómputo e Internet.

2.6 Metodología

1. Las actividades de aprendizaje que se consideran en esta primera parte de la práctica es la solución de los siguientes ejercicios:

1.1 Obtenga en notación currificada las siguientes funciones:

$$inc(x) = x + 1.0$$
$$f(x, y) = x + (4.0 + x)$$

$$doble(x) = x + x$$
$$cuadruple(x) = doble(doble(x))$$

1.2 Asuma que se tienen las siguientes funciones:

$$inc(x) = x + 1.0$$
$$f(x, y) = x + (4.0 + x)$$

Sabiendo que si se intenta reducir una expresión de la forma $^{x}/_{0.0}$, donde x es cualquier número real, se produce un error y la reducción se termina, reduzca la expresión $f(inc\ 5.0)\ (^{7.0}/_{0.0})$ utilizando el orden aplicativo, el orden normal y evaluación perezosa.

1.3 Sean las siguientes definiciones de funciones:

$$doble(x) = x + x$$
$$cuadruple(x) = doble(doble(x))$$

Reduzca la expresión $cuadruple(1 + 2)$ utilizando el orden aplicativo, el orden normal y evaluación perezosa.

2. Las actividades de aprendizaje que se consideran para la segunda parte de la práctica son las siguientes:

2.1 GHC toma un script de Haskell (normalmente tienen la extensión .hs) y lo compila, pero también tiene un modo interactivo el cual nos permite interactuar con dichos scripts. Podemos llamar a las funciones de los scripts que hayamos cargado y los resultados serán mostrados de forma inmediata. Para aprender es mucho más fácil y rápido en lugar de tener que compilar y ejecutar los programas una y otra vez. El modo interactivo se ejecuta tecleando ghci desde la terminal. Tienes que obtener algo similar a la figura 3.

Figura 3. Ejecución del modo interactivo de Haskell

2.2 Usando el interprete de Haskell realice las siguientes operaciones y analice sus resultados:

a) 20*30-5000
b) 20*(30-5000)
c) (20*30)-5000
d) True && False
e) False && False
f) True || True
g) not False
h) == 3
i) = 3
j) “ith” == “ith”
k) “ith” /= “tec”
l) “ith” = “ith”

2.7 Sugerencias didácticas

Después de presentar ejemplos de la evaluación de expresiones y de la construcción de funciones, proponer una serie de ejercicios para la evaluación de expresiones y los alumnos realicen una comparación con otros lenguajes de programación.

2.8 Evidencias de aprendizaje (Reporte del alumno)

Reporte de la práctica de acuerdo al Anexo A.

2.9 Bibliografía

Kurt Will, (2018), Get Programming with Haskell , Manning Publications.

Allen Christopher, Moronuki Julie, (2016), Haskell Programming: From First Principles, Gumroad.

Thomasson Samuli, (2016), Haskell High Perfomance Programming, Packt Publishing Limited.

Church James, Learning Haskell Data Analysis, Packt Publishing Limited.

Bird Richard, (2014), Thinking Functionally with Haskell, Cambridge University Press.

Práctica 3: Tipos predefinidos de la programación funcional

3.1 Competencias a desarrollar

Identificar los elementos de la programación funcional y aplicar la programación funcional.

Habilidad para buscar y analizar información proveniente de fuentes diversas.

Solución de problemas.

Capacidad de aplicar los conocimientos en la práctica.

Habilidades de investigación.

Capacidad de aprender.

3.2 Objetivo de la Práctica

Conocer y aplicar los tipos de datos predefinidos simples y sus funciones en Haskell.

3.3 Introducción

Algunas estructuras de datos son primitivas, es decir, no se derivan de otras estructuras, y son llamadas comúnmente tipos de datos primitivos. Un tipo de datos compuesto está constituido de uno o más tipos de datos primitivos. Los lenguajes de programación proveen un soporte para estos tipos de datos.

Figura 4. Tipos de datos

Como se muestra en la figura 4 los tipos de datos primitivos son el Entero, Booleano, Carácter y Real. El tipo de datos compuesto más común es la cadena de caracteres.

Los lenguajes de programación también se han clasificado por la forma en la que tratan con el problema de los tipos de datos:

- Dinámica y estáticamente tipados
- Fuerte y débilmente tipados

En un lenguaje **estáticamente tipado**, cada variable debe ser declarada con un tipo. Eso ocurre por ejemplo en C# o Java. Para utilizar una variable de tipo entero, es necesario indicar que en efecto, es de ese tipo. El tipado estático es típico de los lenguajes compilados. El tipo no se altera hasta que la variable se sale de ámbito y se extingue. Éste comportamiento se aplica tanto a variables locales (a un método o función), a parámetros (de un método o función), a varibles de instancia (de un objeto), a variables de clase ("static"), o a variables globales (en lenguajes no orientados a objetos).

En un lenguaje **dinámicamente tipado**, no suele ser necesario declarar el tipo de las variables. Es típico de los intérpretes. Ocurre en lenguajes como PHP o Python. Las variables empiezan a existir cuando se les da valor. Los lenguajes dinámicamente tipados tienen en cuenta los tipos, desde luego, pero no de las variables, sino de su contenido. Una misma variable puede contener en un instante dado un dato de un determinado tipo, y en otro instante, puede contener otro dato de otro tipo.

Si un lenguaje nos permite declarar un método o función que acepte como parámetro un número real de doble precisión (es decir, un double), consideraríamios que el lenguaje está muy **fuertemente tipado** si sólo permitiese llamadas o invocaciones a esa función o método pasándole como parámetro un dato con un tipo double.

Por otro lado, casi todos los lenguajes permiten una serie de conversiones implícitas. Es decir, muchos lenguajes (como C# o Java, que son estáticamente tipados) permitirían invocar un método que aceptase un double como parámetro pasándole un float (un número con decimales de simple precisión), o incluso un entero. Es decir, en ese sentido, C# o Java están fuertemente tipados, pero no todo lo fuertemente tipado que se puede estar.

Un lenguaje **débilmente tipado**, en general, permite conversiones implícitas inseguras. Es su filosofía, que el programador se encargue de programar, y el lenguaje se encargará de hacer todo lo posible para que el programa se ejecute, liberando al programador de llevar también la cuenta de posibles conversiones de tipos. Por ejemplo, javascript es un lenguaje con un sistema de tipos mucho más débil que C# o Java. En una función de javascript ni siquiera se declaran los tipos de los parámetros o las variables, ya que el lenguaje es *dinámicamente tipado*.

El lenguaje Haskell está *estáticamente tipado* y se tienen los siguientes tipos de datos predefinidos simples:

El tipo Bool

El tipo Int

El tipo Integer

El tipo Float

El tipo Double

El tipo Char

3.4 Correlación con el o los temas y subtemas del programa de estudio vigente

Tema: Conceptos Fundamentales

Subtema: Disciplina de tipos y tipos de datos.

3.5 Material y equipo necesario

Equipo de cómputo e Internet.

3.6 Metodología

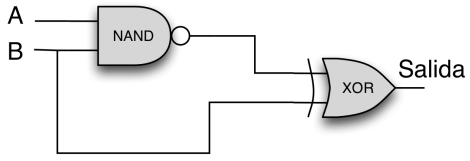
Las actividades de aprendizaje que se consideran en esta práctica es la solución de los siguientes ejercicios:

1. Realice una función en Haskell que obtenga el resultado para la operación lógica OR exclusivo que se muestra en la siguiente tabla verdad:

Tabla Verdad OR Exclusivo

A	B	A orExclusivo B
False	False	False
False	True	True
True	False	True
True	True	False

2. Simule el siguiente circuito usando una función en Haskell.
3. Escriba una función con tipo de dato Int que permita elevar un número grande tal como 9776543212 a la potencia 5.



4. Escriba una función con tipo de dato Integer que permita elevar un número grande tal como 9776543212 a la potencia 5.
5. Analice los resultados de las anteriores funciones y explique tales resultados.
6. En este momento en algunas partes del planeta se está realizando la codificación de algún mensaje para enviarlo por la Internet. Asuma que el sistema de criptografía usado es el RSA (Rivest, Shamir y Adleman), enseguida se muestra un ejemplo del algoritmo. Construya un par de funciones para implementar encrypt y decrypt en Haskell.

Aquí tenemos un ejemplo de cifrado/descifrado con RSA. Los parámetros usados aquí son pequeños y orientativos con respecto a los que maneja el algoritmo, pero podemos usar también OpenSSL para generar y examinar un par de claves reales.

$p = 61$ 1º nº primo privado

$q = 53$ 2º nº primo privado

$n = pq = 3233$ producto $p \times q$

$e = 17$ exponente público

$d = 2753$ exponente privado

La clave pública (e, n). La clave privada es (d, n). La función de cifrado es:

$$\text{encrypt}(m) = m^e \pmod{n} = m^{17} \pmod{3233}$$

Donde m es el texto sin cifrar. La función de descifrado es:

$$\text{decrypt}(c) = c^d \pmod{n} = c^{2753} \pmod{3233}$$

Donde c es el texto cifrado. Para cifrar el valor del texto sin cifrar 123, nosotros calculamos:

$$\text{encrypt}(123) = 123^{17} \pmod{3233} = 855$$

Para descifrar el valor del texto cifrado, nosotros calculamos:

$$\text{decrypt}(855) = 855^{2753} \pmod{3233} = 123$$

7. En el procesamiento de imágenes de rostros, una de las tareas es realizar la rotación del rostro de tal manera que se obtenga un línea horizontal al nivel de los ojos. Por lo tanto es necesario obtener las posiciones de los ojos y a través de geometría se puede obtener las nuevas posiciones para efectuar la rotación. En las siguientes figuras se muestra este procedimiento.

Escriba las funciones en Haskell que sean necesarias para realizar el proceso descrito previamente y acuerdo a las ecuaciones que se muestran a continuación para obtener el ángulo de rotación y las nuevas posiciones de los ojos:

Donde yder, yizq, xder y xizq son las posiciones de los ojos.

angulo = -atan((yder –yizq) / (xder –xizq))

x0 = (xder + xizq) / 2

y0 = (yder + yizq) / 2

COS = cos(-angulo)

SIN = sin(-angulo)

Nuevos valores del ojo izquierdo

xxizq = COS * (xizq –x0) – SIN * (yizq – y0) + x0

yyizq = SIN * (xizq –x0) + COS * (yizq –y0) + y0

Nuevos valores del ojo derecho

xxder = COS * (xder –x0) – SIN * (yder – y0) + x0

yyder = SIN * (xder –x0) + COS * (yder –y0) + y0

3.7 Sugerencias didácticas

Presentar a los alumnos los diferentes tipos de datos, posteriormente discutir con los alumnos los diferentes formatos usados para los tipos de datos en la programación. Cuestionar el uso de cada tipo de acuerdo a su aplicación.

3.8 Evidencias de aprendizaje (Reporte del alumno)

Reporte de la práctica de acuerdo al Anexo A.

3.9 Bibliografía

Rose-Gómez César Enrique, (2012), Administración y Organización de Datos, Tecnológico Nacional de México / Instituto Tecnológico de Hermosillo.

Kurt Will, (2018), Get Programming with Haskell , Manning Publications.

Allen Christopher, Moronuki Julie, (2016), Haskell Programming: From First Principles, Gumroad.

Thomasson Samuli, (2016), Haskell High Perfomance Programming, Packt Publishing Limited.

Church James, Learning Haskell Data Analysis, Packt Publishing Limited.

Bird Richard, (2014), Thinking Functionally with Haskell, Cambridge University Press.

Programación Funcional

Apuntes y Prácticas

Práctica 4: Patrones de la programación funcional

4.1 Competencias a desarrollar

Identificar los elementos de la programación funcional y aplicar la programación funcional.

Habilidad para buscar y analizar información proveniente de fuentes diversas.

Solución de problemas.

Capacidad de aplicar los conocimientos en la práctica.

Habilidades de investigación.

Capacidad de aprender.

4.2 Objetivo de la Práctica

Conocer y aplicar los tipos de patrones y sus funciones en Haskell.

4.3 Introducción

Los lenguajes de programación generalmente proporcionan a los desarrolladores muy pocos tipos primitivos, pero también proporcionan los tipos construidos o estructurados predefinidos para representar colecciones, estos tipos tienen asociados operadores o funciones para su uso.

Las tuplas es uno de los tipos más importantes de tipos construidos. Conceptualmente, las tuplas son productos cartesianos de otros tipos. Si se consideran los tipos como dominios, entonces el producto cartesiano de dominios $D_1, D_2, \ldots, D_k$, escrito como: $D_1 x D_2 x \ldots x D_k$ es el conjunto de todas las k tuplas $(v_1, v_2, \ldots, v_k)$ tal que v_1 está en D_1, v_2 está en D_2, etc.

Por ejemplo: Si tenemos k=2, $D_1 = \{0,1\}$, y $D_2 = \{a,b,c\}$, entonces: $D_1 x D_2 = \{(0,a),(0,b),(0,c),(1,a),(1,b),(1,c)\}$;

De tal manera que una tupla en un lenguaje funcional es un dato compuesto donde cada elemento puede ser de un tipo diferente. Si $v_1, v_2, v_3, \dots, v_n$ son valores con tipo $t_1, t_2, t_3, \dots, t_n$ entonces $(v_1, v_2, v_3, \dots, v_n)$ es una tupla con tipo $(t_1, t_2, t_3, \dots, t_n)$.[1]

Otro tipo construido también muy importante es la cadena de caracteres, la cual esta definida como una secuencia de cero o más caracteres. En Haskell se tiene definida como se muestra a continuación:

$$"c_1 c_2 c_3 \dots c_{n-1} c_n" \leftrightarrow ['c_1', 'c_2', 'c_3', \dots, 'c_{n-1}', 'c_n']$$

4.4 Correlación con el o los temas y subtemas del programa de estudio vigente

Tema: Programación funcional

Subtema: Tipos de datos, funciones y operadores.

4.5 Material y equipo necesario

Equipo de cómputo e Internet.

4.6 Metodología

Las actividades de aprendizaje que se consideran en esta práctica es la solución de los siguientes ejercicios:

1. Construya una función que tenga como parámetro un carácter y regrese una tupla con tres valores: el carácter, su código ASCII y si es mayúscula a través de un valor booleano. Es importante recordar que para el uso de algunas funciones para el manejo de datos Char es necesario importar la librería a través de la siguiente sentencia: import Data.Char

2. Para un modelo de abarrotes, se requiere mantener en una tupla el nombre del producto y el precio del producto, de tal manera que se necesitamos funciones que nos permitan obtener el nombre del producto o el precio del producto. Para ello considere que una cadena de caracteres en Haskell es establecida con comillas dobles, por ejemplo; "soda". Asimismo considere que para obtener los elementos de una tupla se pueden usar las funciones fst y snd que son selectores de pares.

4.7 Sugerencias didácticas

Presentar a los alumnos los diferentes tipos de datos complejos, posteriormente discutir con los alumnos los diferentes formatos complejos definidos por los usuarios.

4.8 Evidencias de aprendizaje (Reporte del alumno)

Reporte de la práctica de acuerdo al Anexo A.

4.9 Bibliografía

Kurt Will, (2018), Get Programming with Haskell , Manning Publications.

Allen Christopher, Moronuki Julie, (2016), Haskell Programming: From First Principles, Gumroad.

Thomasson Samuli, (2016), Haskell High Perfomance Programming, Packt Publishing Limited.

Church James, Learning Haskell Data Analysis, Packt Publishing Limited.

Bird Richard, (2014), Thinking Functionally with Haskell, Cambridge University Press.

Programación Funcional

Apuntes y Prácticas

Práctica 5: Recursividad en la programación funcional

5.1 Competencias a desarrollar

Identificar los elementos de la programación funcional y aplicar la programación funcional.

Habilidad para buscar y analizar información proveniente de fuentes diversas.

Solución de problemas.

Capacidad de aplicar los conocimientos en la práctica.

Habilidades de investigación.

Capacidad de aprender.

5.2 Objetivo de la Práctica

Conocer y aplicar la recursividad en Haskell.

5.3 Introducción

Cuando escribimos un método para resolver un problema en particular, una de las técnicas básicas de diseño es dividir la tarea en subtareas más pequeñas. Por ejemplo, el problema de agregar (o multiplicar) n enteros consecutivos se puede reducir a un problema de agregar (o multiplicar) n-1 enteros consecutivos:

```
1 + 2 + 3 +... + n = n + [1 + 2 + 3 + .. + (n-1)]

1 * 2 * 3 *... * n = n * [1 * 2 * 3 * .. * (n-1)]
```

Por lo tanto, si introducimos un método *sumaR(n)* (o *vecesR(n)*) que agrega (o multiplica) enteros de *1* a *n*, entonces las expresiones aritméticas anteriores pueden reescribirse como:

```
sumaR(n) = n + sumaR(n-1)
vecesR(n) = n * vecesR(n-1)
```

Dicha definición funcional se denomina definición recursiva, ya que la definición contiene una llamada a sí misma. En cada llamada recursiva, el argumento de *sumaR(n)* (o *vecesR(n)*) se reduce en uno. Lleva n-1 llamadas hasta que lleguemos al

caso base; esto es parte de una definición que no hace una llamada a sí mismo. Cada definición recursiva requiere casos base para evitar la recursión infinita.

La programación recursiva está directamente relacionada con la inducción matemática El caso base es demostrar que la afirmación es cierta para algunos valores o valores específicos de N. El paso de inducción: suponga que una declaración es verdadera para todos los enteros positivos menores que N, luego demuestre que es verdadera para N.

En matemáticas y también en la computación nos enfrentamos a la solución de problemas que por su naturaleza tienen una definición recursiva, por ejemplo; el calculo del factorial de un numero entero o la definición de un árbol binario, entre otros.

Particularmente en la programación declarativa el mecanismo de la recursividad es muy utilizado ya que no se tiene la estructura iterativa, de tal manera que el problema se resuelve a través de la recursividad.

Un algoritmo es llamado recursivo si resuelve un problema reduciéndolo a una instancia del mismo problema con una entrada pequeña. Un ejemplo clásico de la recursividad es el cálculo del factorial de un número entero. Esta función esta definida como:

$$n! = \begin{cases} 1 \; si \; n=1 \\ (n*(n-1)! \; si \; n>1 \end{cases}$$

La función del cálculo del factorial puede ser escrita como:

entero Factorial(n)

if (n==1) entonces

regresa 1

de otra manera

*regresa (n*Factorial(n-1))*

La recurrencia del factorial del número 3 es la siguiente:

3! = 3 * (3-1)!

= 3 * 2!

= 3 * 2 * (2-1)!

= 3 * 2 * 1!

= 3 * 2 * 1 * (1-1)!

= 3 * 2 * 1 * 0!

= 3 * 2 * 1 * 1

= 6

Si la llamada recursiva se produce al final de un método, se denomina recursividad final. La recursividad de cola es similar a un lazo. El método ejecuta todas las instrucciones antes de pasar a la siguiente llamada recursiva. Si la llamada recursiva se produce al inicio de un método, se llama recursividad principal. El método guarda el estado antes de saltar a la siguiente llamada recursiva.

5.4 Correlación con el o los temas y subtemas del programa de estudio vigente

Tema: Programación funcional

Subtema: Funciones.

5.5 Material y equipo necesario

Equipo de cómputo e Internet.

5.6 Metodología

Las actividades de aprendizaje que se consideran en esta práctica es la solución de los siguientes ejercicios:

1. Escriba un programa en Haskell para obtener el factorial de un numero entero.
2. Fibonacci nació en el año 1170 en Pisa, Italia, y murió en 1250. Su verdadero nombre fue Leonardo Pisano. En 1202, escribió un libro: Liber Abbaci, que significa "Libro de cálculo".

   El número de Fibonacci se define como la suma de los dos números precedentes:

   0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

   Escriba un programa en Haskell para obtener la secuencia Fibonacci de longitud *n*.

   Escriba un programa en Haskell que calcule el resto de una división de numero enteros. Por ejemplo, una llamada *resdiv(100,12)* regresará el valor 4 (debido a que 100 dividido por 12 es igual a 8 con resto 4).
3. Escriba un programa en Haskell que calcule la división de dos números enteros usando restas sucesivas.

5.7 Sugerencias didácticas

Presentar a los alumnos el concepto de recursividad desde el punto de vista matemático. Asociar el concepto de recursividad con aspectos computacionales como las llamadas a procedimientos dentro de la gestión de procesos del sistema operativo.

5.8 Evidencias de aprendizaje (Reporte del alumno)

Reporte de la práctica de acuerdo al Anexo A.

5.9 Bibliografía

Kurt Will, (2018), Get Programming with Haskell , Manning Publications.

Allen Christopher, Moronuki Julie, (2016), Haskell Programming: From First Principles, Gumroad.

Thomasson Samuli, (2016), Haskell High Perfomance Programming, Packt Publishing Limited.

Church James, Learning Haskell Data Analysis, Packt Publishing Limited.

Bird Richard, (2014), Thinking Functionally with Haskell, Cambridge University Press.

Tanenbaum, A.S., (2003), Sistemas Operativos Modernos, 3ED, Pearson Educación.

Práctica 6: Estructuras de datos lineales en la programación funcional

6.1 Competencias a desarrollar

Identificar los elementos de la programación funcional y aplicar la programación funcional.

Habilidad para buscar y analizar información proveniente de fuentes diversas.

Solución de problemas.

Capacidad de aplicar los conocimientos en la práctica.

Habilidades de investigación.

Capacidad de aprender.

6.2 Objetivo de la Práctica

Conocer y aplicar los diferentes constructores de tipos predefinidos en Haskell para su uso en listas.

6.3 Introducción

Podemos definir lo que es estructura de datos de dos maneras distintas, aunque complementarias:

1. Es la forma en que están relacionados objetos de datos o datos complejos.
2. Es una clase que puede ser caracterizada por su organización y por las operaciones susceptibles de realizar con tales datos.

En la figura 5 se categorizan las diferentes estructuras de datos. Algunas son primitivas, es decir, no se derivan de otras estructuras, y son llamadas comúnmente tipos de datos primitivos. Un tipo de datos compuesto está constituido de uno o más tipos de datos primitivos. Los lenguajes de programación proveen un soporte para estos tipos de datos.

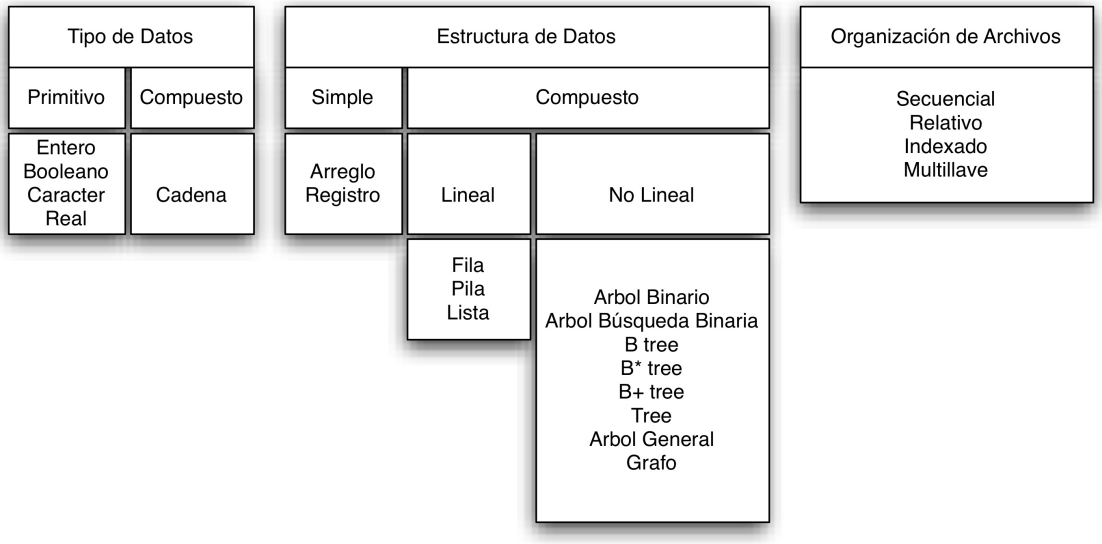


Figura 5. Estructuras de Datos

Los tipos de datos pueden ser organizados en varias formas para establecer las estructuras de datos simples y estas estructuras pueden ser combinadas para formas estructuras más complejas, algunas veces llamadas estructuras de datos compuestas.

Hay dos tipos fundamentales de estructuras de datos complejas: lineales y no lineales, dependiendo de la complejidad de las relaciones lógicas que las representan. Las estructuras de datos aplicadas a colecciones de datos en almacenamiento secundario son llamadas organizaciones de archivo.

Los datos pueden ser representados, organizados y almacenados de la forma más eficiente posible, lo cual dependerá obviamente de la aplicación y de la selección de la estructura de datos más adecuada, misma que se transformará según la complejidad que se requiera.

Una de las estructuras lineales, son las listas. Las cuales constituyen una estructura flexible en particular, porque puede crecer y acortarse según se requiera, los elementos son accesibles y se pueden insertar y suprimir en cualquier posición de la lista, considerando su propiedad lineal, esto es, se tiene que recorrer linealmente para realizar las operaciones mencionadas anteriormente.

Las lista también pueden concatenarse entre sí o dividirse en sublistas; se presentan de manera rutinaria en aplicaciones como recuperación de información, traducción de lenguajes de programación, simulación, entre otros.

Matemáticamente, una lista es una secuencia de cero o más elementos de tipo determinado:

$$a_1, a_2, a_3, \ldots \ldots \ldots, a_n$$

donde $n \geq 0$, a_i es el elemento i y n es la longitud de la lista.

Al asumir que $n \geq 1$, se dice que a_1 es el primer elemento y a_n es el último elemento. Si $n = 0$ se tiene una lista vacía, es decir, que no tiene elementos.

Una propiedad importante de una lista es que sus elementos están ordenados en forma lineal de acuerdo a sus posiciones en la misma, se dice que a_i esta en la posición i.

6.4 Correlación con el o los temas y subtemas del programa de estudio vigente

Tema: Programación funcional

Subtema: Aplicaciones de las listas.

6.5 Material y equipo necesario

Equipo de cómputo e Internet.

6.6 Metodología

Las actividades de aprendizaje que se consideran en esta práctica es la solución de los siguientes ejercicios:

1. Usando la interfaz del interprete en modo interactivo realice las siguientes expresiones y analice sus resultados:

   Prelude> let numeros = [1,2,3,4]

   Prelude> let verdad = [True, False, False]

   Prelude> let cadenas = ["aqui", "estan", "algunas", "cadenas"]

   Prelude> let mezcla = [True, "hola"]

2. Escriba una función en Haskell que sume todos los elementos de una lista de números.
3. Dada una lista de caracteres, escriba una función en Haskell que regrese la lista de sus ordinales.
4. Escriba una función en Haskell que calcule la lista de los restos de la división de los elementos de una lista de números dada por otro número dado.
5. Dada una lista de números, escriba una función en Haskell que regrese la lista de sus cuadrados.
6. Dada una lista de listas, escriba una función en Haskell que regrese la lista de sus longitudes.
7. Dada una lista de enteros, escriba una función en Haskell que regrese la lista de los elementos pares.
8. Usando listas por comprensión, define una función:
9. expandir :: [Integer] -> [Integer] que reemplace en una lista de números positivos cada número n por n copias de sí mismo, por ejemplo:
10. ? expandir [1::4]
11. [1,2,2,3,3,3,4,4,4,4] :: [Integer]

12. Escriba un programa en Haskell para contar cuantos elementos pares hay en una lista. Estamos diciendo que *x* pertenece a la lista y además debe cumplir la condición de ser par. Como en varios lenguajes la función *length* cuenta los elementos.
13. Escriba un programa en Haskell para regresar los cuadrados de una lista que contenga los primeros 50 enteros. Considere la lista por comprensión.
14. Escriba un programa en Haskell para regresar una lista de números primos de 1 a *n*. Para ello debemos crear nuestra función para saber si un numero es primo o no y después la aplicamos a la lista por comprensión.

6.7 Sugerencias didácticas

Analizar con los alumnos las diferencias entre estructuras de datos lineales y no lineales. Asimismo, cuales son las aplicaciones de las estructuras de datos lineales.

6.8 Evidencias de aprendizaje (Reporte del alumno)

Reporte de la práctica de acuerdo al Anexo A.

6.9 Bibliografía

Kurt Will, (2018), Get Programming with Haskell , Manning Publications.

Allen Christopher, Moronuki Julie, (2016), Haskell Programming: From First Principles, Gumroad.

Thomasson Samuli, (2016), Haskell High Perfomance Programming, Packt Publishing Limited.

Church James, Learning Haskell Data Analysis, Packt Publishing Limited.

Bird Richard, (2014), Thinking Functionally with Haskell, Cambridge University Press.

Práctica 7: Expresiones condicionales de la programación funcional

7.1 Competencias a desarrollar

Conocer la evaluación perezosa.

Identificar la evaluación perezosa como una de las funciones de la programación representativa.

Aplicar la evaluación perezosa en la resolución de problemas.

Conocer las expresiones condicionales de la programación funcional.

Diseñar programación con modularidad.

Solución de problemas.

Capacidad de aplicar los conocimientos en la práctica.

Habilidades de investigación.

Capacidad de aprender.

7.2 Objetivo de la Práctica

Conocer y aplicar las expresiones condicionales en Haskell.

7.3 Introducción

La forma general de un estatuto condicional que engloba todos los constructores condicionales es el estatuto *guarded if* introducido por E.W. Disjkstra, el cual se muestra a continuación:

$$
\begin{aligned}
&if\ \ B1\ \rightarrow S1\\
&\quad |\ B2\ \rightarrow S2\\
&\quad |\ B3\ \rightarrow S3\\
&\qquad \ldots\ldots\\
&\quad |\ Bn\ \rightarrow Sn\\
&fi
\end{aligned}
$$

La semántica de este estatuto es la siguiente: las Bi′s son todas las expresiones booleanas, llamadas guardas y las Si′s son estatutos secuencia. Si una de las Bi′s evalúa a true, entonces el estatuto secuencia correspondiente Si es ejecutado. Si más de uno de los Bi′s es true, entonces uno y solo uno de los correspondientes Si′s es seleccionado para su ejecución. Si ninguna de las Bi′s es true, entonces un error ocurre.

Aquí no se establece que el primer Bi que evalúa a true es el elegido, por lo tanto el estatuto guarded if introduce el no determinismo en la programación (muy útil en la programación concurrente). Asimismo, deja sin especificar si todos los guardas son evaluados.

7.4 Correlación con el o los temas y subtemas del programa de estudio vigente

Tema: Evaluación perezosa

Subtema: La estrategia de evaluación perezosa y técnicas de programación funcional perezosa.

7.5 Material y equipo necesario

Equipo de cómputo e Internet.

7.6 Metodología

Las actividades de aprendizaje que se consideran en esta práctica es la solución de los siguientes ejercicios:

1. Escribe una función que determine el menor de tres números enteros.
2. Usando if-then-else escriba la definición de:

$$|x| = x \; si \; x \geq 0$$
$$|x| = |-x| \; si \; x < 0$$

3. Usando guardas escriba la definición de:

$$|x| = x \; si \; x \geq 0$$
$$|x| = -x \; si \; x < 0$$

4. Usando guardas y where escriba la definición de la siguiente ecuación:

$$ax^2 + bx + c = 0$$

5. Usando if-then-else y where escriba la definición de la siguiente ecuación:

$$ax^2 + bx + c = 0$$

7.7 Sugerencias didácticas

Considerar la estructura de expresiones condicionales para comprender el concepto de evaluación perezosa.

7.8 Evidencias de aprendizaje (Reporte del alumno)

Reporte de la práctica de acuerdo al Anexo A.

7.9 Bibliografía

Kurt Will, (2018), Get Programming with Haskell , Manning Publications.

Allen Christopher, Moronuki Julie, (2016), Haskell Programming: From First Principles, Gumroad.

Thomasson Samuli, (2016), Haskell High Perfomance Programming, Packt Publishing Limited.

Church James, Learning Haskell Data Analysis, Packt Publishing Limited.

Bird Richard, (2014), Thinking Functionally with Haskell, Cambridge University Press.

Práctica 8: Estructuras de datos no lineales en la programación funcional

8.1 Competencias especificas de la unidad

Diseñar programación con modularidad.

Aplicar las estructuras de datos no lineales.

Solución de problemas.

Capacidad de aplicar los conocimientos en la práctica.

Habilidades de investigación.

Capacidad de aprender.

8.2 Objetivo de la Práctica

Conocer y aplicar los diferentes constructores de tipos predefinidos en Haskell para su uso en árboles.

8.3 Introducción

Podemos definir lo que es estructura de datos de dos maneras distintas, aunque complementarias:

- Es la forma en que están relacionados objetos de datos o datos complejos.
- Es una clase que puede ser caracterizada por su organización y por las operaciones susceptibles de realizar con tales datos.

En la figura 6 se categorizan las diferentes estructuras de datos. Algunas son primitivas, es decir, no se derivan de otras estructuras, y son llamadas comúnmente tipos de datos primitivos. Un tipo de datos compuesto está constituido de uno o más tipos de datos primitivos. Los lenguajes de programación proveen un soporte para estos tipos de datos.

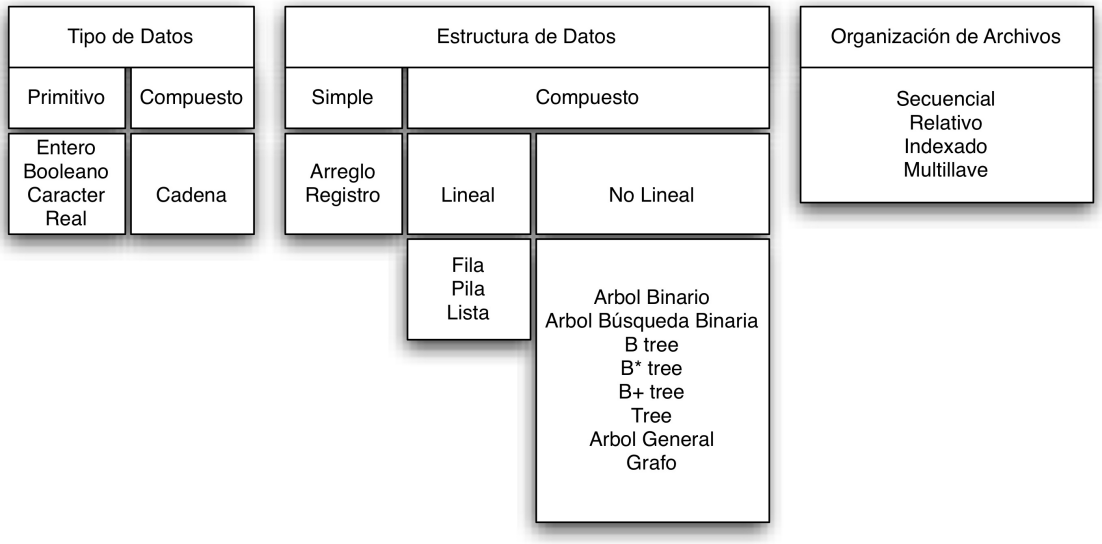


Figura 6. Estructuras de Datos

Los tipos de datos pueden ser organizados en varias formas para establecer las estructuras de datos simples y estas estructuras pueden ser combinadas para formas estructuras más complejas, algunas veces llamadas estructuras de datos compuestas.

Hay dos tipos fundamentales de estructuras de datos complejas: lineales y no lineales, dependiendo de la complejidad de las relaciones lógicas que las representan. Las estructuras de datos aplicadas a colecciones de datos en almacenamiento secundario son llamadas organizaciones de archivo.

Los datos pueden ser representados, organizados y almacenados de la forma más eficiente posible, lo cual dependerá obviamente de la aplicación y de la selección de la estructura de datos más adecuada, misma que se transformará según la complejidad que se requiera.

Una de las estructuras no lineales, son los árboles. Un árbol impone una estructura jerárquica sobre una colección de objetos. Los árboles genealógicos y los organigramas son ejemplos comunes de árboles. Entre otras cosas, los árboles son útiles para analizar circuitos eléctricos y para representar la estructura de formulas matemáticas. También se presentan naturalmente en diversas áreas de computación, entre las cuales se tienen:

i. Para organizar tablas de símbolos en ensambladores, compiladores, etc.
ii. Para el análisis sintáctico en compiladores y sistemas que tengan lenguajes de comandos.
iii. Se usan en algunos métodos de búsqueda y ordenamiento.
iv. En la construcción de espacios para la representación de tablas de decisión, en teorías de juegos, en la prueba automática de teoremas, etc.

v. En el manejo de archivos para el acceso a través de índices.

Un árbol es una colección de elementos llamados nodos, de los cuales uno se distingue como raíz, junto con una relación de paternidad, que impone una estructura jerárquica sobre los nodos. Un nodo, como un elemento de una lista, puede ser del tipo que se desee.

Definición recursiva:

1. Un solo nodo es, por si mismo, un árbol. Ese nodo es también la raíz de dicho árbol.
2. Asuma que n es un nodo y que $A_1, A_2, \ldots, A_k$ subárboles con raíces $n_1, n_2, \ldots, n_k$, respectivamente. Se puede construir un nuevo árbol haciendo que n se constituya en el padre de los nodos $n_1, n_2, \ldots, n_k$. En dicho árbol, n es la raíz de $A_1, A_2, \ldots, A_k$ que son subárboles de la raíz. Los nodos $n_1, n_2, \ldots, n_k$ reciben el nombre de hijos del nodo n.

Considere el árbol que se muestra en la figura 7, el cual es un árbol general.

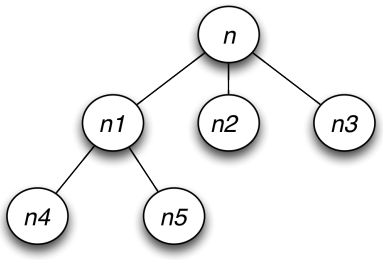


Figura 7. Árbol general

En este árbol notamos que *n* es la raíz del árbol, *n1* es un antecesor de *n4* y *n4* es descendiente de *n1*. La longitud de un camino es el numero de nodos del camino menos 1, hay un camino de longitud cero de cualquier nodo a sí mismo.

La altura de un nodo en un árbol es la longitud del camino mas largo de ese nodo a una hoja, la altura del árbol es la altura de la raíz. La profundidad de un nodo es la longitud del camino único desde la raíz a ese nodo.

Normalmente los hijos de un nodo se ordenan de izquierda a derecha, pero si se desea explícitamente ignorar el orden de los hijos, se dice que el árbol es no ordenado.

Existen varias maneras de ordenar sistemáticamente todos los nodos de un árbol. Los tres ordenamientos mas importantes son el orden previo (preorden), orden simétrico (inorden) y orden posterior (postorden).

Un árbol binario es un conjunto de nodos, cuyo grado máximo del nodo es igual a 2, por lo tanto pueden estar vacíos o consiste de dos árboles binarios llamados subárboles izquierdo y derecho.

8.4 Correlación con el o los temas y subtemas del programa de estudio vigente

Tema: Programación funcional

Subtema: Aplicaciones de los árboles.

8.5 Material y equipo necesario

Equipo de cómputo e Internet.

8.6 Metodología

Las actividades de aprendizaje que se consideran en esta práctica es la solución de los siguientes ejercicios:

1. Considerar la siguiente definición de tipo para representar árboles generales:

   data GenTree a = Gen a [GenTree a]

   y la función de reducción para estos árboles, definida como sigue:

   foldGen :: (a → b → c) → ([y] → b) → Gentree a → c

   foldGen f g (Gen label hijos) = f label (g (map (foldGen f g) hijos))

   Usando *foldGen*, definir y dar el tipo de las siguientes funciones, e implementarlas en Haskell:

   a) *sumGen*, que devuelve la suma de todos los elementos de un árbol de enteros.

   b) *doble*, que multiplica por dos todos los elementos de un árbol de enteros;

   c) *mapGen*, que es análoga a la función *map* de listas;

   d) *mirrorGen,* es el árbol obtenido intercambiando los subárboles izquierdo y derecho de cada nodo.

2. Dado el tipo:
   data Bintree a = Empty |Bin a (binTree a) (BinTree a)
   a) Definir la función
   foldBin :: (a → b → b → b) → b → BinTree a → b
   b) Usando *foldBin* , definir la función *mapBin*
   c) Definir las funciones *heighBin* y *mirrorBin*, usando *foldBin*

8.7 Sugerencias didácticas

Analizar con los alumnos las diferencias entre estructuras de datos lineales y no lineales. Asimismo, cuales son las aplicaciones de las estructuras de datos no lineales.

8.8 Evidencias de aprendizaje (Reporte del alumno)

Reporte de la práctica de acuerdo al Anexo A.

8.9 Bibliografía

Kurt Will, (2018), Get Programming with Haskell , Manning Publications.

Allen Christopher, Moronuki Julie, (2016), Haskell Programming: From First Principles, Gumroad.

Thomasson Samuli, (2016), Haskell High Perfomance Programming, Packt Publishing Limited.

Church James, Learning Haskell Data Analysis, Packt Publishing Limited.

Bird Richard, (2014), Thinking Functionally with Haskell, Cambridge University Press.

Práctica 9: Otros tipos de datos en la programación funcional

9.1 Competencias especificas de la unidad.

Resolución de problemas que requieren de tipos de datos complejos o abstractos.

Diseñar programación con modularidad.

Solución de problemas.

Capacidad de aplicar los conocimientos en la práctica.

Habilidades de investigación.

Capacidad de aprender.

9.2 Objetivo de la Práctica

Conocer y aplicar los diferentes constructores de tipos en Haskell.

9.3 Introducción

El mecanismo para construir tipos de datos adicionales a los datos primitivos o compuestos en los lenguajes de programación, debe proporcionar lo siguiente:

Un método para definir un tipo de dato y las operaciones necesarias para ese tipo de dato. Las definiciones deben estar en un solo lugar y las operaciones deben estar directamente asociadas a ese tipo de dato. Las definiciones no deben depender de detalles de la implementación. Las definiciones de las operaciones deben incluir una especificación de sus semánticas.

Un método que incluya los detalles de la implementación del tipo y sus operaciones en solo lugar, restringir el acceso a estos detalles por programas que usen el tipo de dato.

Un tipo de dato construido usando el mecanismo satisfaciendo alguno o todos los criterios previamente descritos es frecuentemente llamado *"tipo de dato abstracto" (TDA).*

Por ejemplo, para un tipo de dato llamado *cola*, una especificación sintáctica es similar a lo siguiente:

type cola(elemento) imports booleano

operaciones:

crear: → cola

agregar: cola x elemento → cola

sacar: cola → cola

frente: cola → elemento

vacia: cola → booleano

variables: q: cola; x: elemento

axiomas:

vacia(crear) = true

vacia(agregar(q,x)) = false

frente(crear) = error

frente(agregar(q,x)) = if vacia(q) then x else frente(q)

sacar(crear) = error

sacar(agregar(q,x)) = if vacia(q) then q else agregar(sacar(q),x)

En Haskell existen una variedad de tipos de datos complejos que ya están construidos y también pueden ser definidos por el usuario. Ejemplos de estos son los sinónimos de tipo, tipos de datos derivados, tipos enumerados, uniones, productos, registros variantes, tipos recursivos y los tipos polimórficos.

9.4 Correlación con el o los temas y subtemas del programa de estudio vigente

Tema: Evaluación perezosa.

Subtema: La estrategia de evaluación perezosa y técnicas de programación funcional perezosa.

9.5 Material y equipo necesario

Equipo de cómputo e Internet.

9.6 Metodología

Las actividades de aprendizaje que se consideran en esta práctica es la solución de los siguientes ejercicios:

1. Considere el siguiente modelo de "base de datos" para prestamos bibliotecarios, asuma que esta base de datos es la lista de prestamos, donde cada préstamo es un par persona-libro.

   type Persona = String

   type Libro = String

   type BD_Bi

   blioteca = [(Persona,Libro)]

   Construya un programa en Haskell que permita realizar las siguientes funcionalidades con las base de datos:
   a. Dada una persona, averiguar los libros que tiene prestados
   b. Dado un libro, averiguar quienes lo tienen prestado
   c. Dada una base de datos, introducir un nuevo préstamo al tomar prestado un libro una persona
   d. Dada una base de datos, eliminar un préstamo

2. Definir un tipo de Colores y una función mezclar que permita la combinación de los mismos, considerando que se tiene el modelo de color RGB, esto es, se combinan tres valores de colores.

3. Se requiere un tipo de dato binario para la solución de cierto problema.
   a. Escriba un programa en Haskell que permita las operaciones de suma y producto módulo 2 para el siguiente tipo de dato:

      data DigBin = Cero | Uno

   b. Asimismo el programa debe tener la definición de las operaciones de suma binaria, producto por dos, cociente y resto de la división por dos para el tipo:

      type NumBin = [Digbin]

      donde convenimos que el primer elemento de las lista de dígitos es el dígito menos significativo del número representado.

   c. Redefinir las funciones del item anterior, observando una convención opuesta.
   d. Definir funciones que multipliquen números binarios de acuerdo a las dos convenciones.
4. Definir las operaciones de unión e intersección de dos conjuntos y el predicado de pertenencia para conjuntos de elementos de tipo *a* representados.
   a. por comprensión (como predicado $a \rightarrow Bool$);
   b. por extensión (como lista de elementos de *a*).

9.7 Sugerencias didácticas

Analizar con los alumnos la implementación de los tipos de datos abstractos de otros lenguajes de programación y sus aplicaciones.

9.8 Evidencias de aprendizaje (Reporte del alumno)

Reporte de la práctica de acuerdo al Anexo A.

9.9 Bibliografía

Kurt Will, (2018), Get Programming with Haskell , Manning Publications.

Allen Christopher, Moronuki Julie, (2016), Haskell Programming: From First Principles, Gumroad.

Thomasson Samuli, (2016), Haskell High Perfomance Programming, Packt Publishing Limited.

Church James, Learning Haskell Data Analysis, Packt Publishing Limited.

Bird Richard, (2014), Thinking Functionally with Haskell, Cambridge University Press.

Programación Funcional

Apuntes y Prácticas

Práctica 10: Funciones polimórficas de la programación funcional

10.1 Competencias especificas de la unidad.

Resolución de problemas que requieren de funciones polimórficas.

Diseñar programación con modularidad.

Solución de problemas.

Capacidad de aplicar los conocimientos en la práctica.

Habilidades de investigación.

Capacidad de aprender.

10.2 Objetivo de la Práctica

Conocer y aplicar los diferentes constructores de funciones polimórficas en Haskell.

10.3 Introducción

Una operación es polimórfica si su argumento puede pertenecer a varios dominios semánticos. La respuesta que produce es dependiente del dominio de su argumento. Un ejemplo de esto, es la operación suma de propósito general, que produce la suma como un entero de dos argumentos de tipo entero y una suma racional de dos argumentos de tipo racional. Esta operación puede definirse funcionalmente como se muestra a continuación:

$$(Integer\ x\ Integer) \cup (Rational\ x\ Rational) \rightarrow Integer \cup Rational$$

El polimorfismo aparece en los lenguajes de programación de propósito general. Se encuentran dos tipos: *polimorfismo ad hoc*, llamado también sobrecarga (*overloading*) y *polimorfismo paramétrico*. El operador del polimorfismo ad hoc se comporta diferentemente para argumentos de diferentes tipos, mientras que la operación polimórfica se comporta igual para todos los tipos.

En algunos lenguajes funcionales como Miranda o en Orwell, las funciones pueden ser monomórficas o polimórficas. En la expresión de tipo de las primeras no

aparecen variables de tipo, mientras que en las segundas debe aparecer alguna variable de tipo. Ejemplos típicos son:

```
(.) :: (b → c) → (a → b) → (a → c)
length :: [a] → Integer
(.) f g x = f (g x )
length [ ] = 0
length ( _ : xs ) = 1 + length xs
```

Las funciones polimórficas son más reutilizables que las monomórficas ya que pueden actuar con cualquier tipo que se obtenga al sustituir las variables de tipo por tipos concretos. Además, la misma función puede actuar con distintos tipos en una expresión. Lo importante es que el código de una función polimórfica no depende de las variables de tipo.

Existe un tercer grupo de funciones que se encuentran entre las funciones monomórficas y las polimórficas: las sobrecargadas. Estas funciones tienen sentido para más de un tipo, pero no para cualquiera. Además, el código de una función sobrecargada puede depender del tipo de alguno de sus argumentos o del resultado.

Un ejemplo simple de función sobrecargada es el operador (+). En ciertos lenguajes (funcionales, lógicos e imperativos) es posible utilizarlo con sobrecarga: 2 + 5, 2.2 + 5.5.

Aunque algunos lenguajes permiten esta sobrecarga, lo que normalmente no está permitido es ampliar la sobrecarga o alterarla, de forma que podamos escribir 2 + 2.3, True + False, o también ‘c‘+ ‘d’.

La solución adoptada por Haskell para modelar la sobrecarga es a través del concepto de clases de tipos. Una clase es un conjunto de tipos para los que tiene sentido un conjunto de funciones. La declaración de las funciones que van a pertenecer a una clase se realiza con una declaración de clase, mientras que para incluir cierto tipo en una clase se utiliza una declaración de instancia.

10.4 Correlación con el o los temas y subtemas del programa de estudio vigente

Tema: Evaluación perezosa.

Subtema: La estrategia de evaluación perezosa y técnicas de programación funcional perezosa.

10.5 Material y equipo necesario

Equipo de cómputo e Internet.

10.6 Metodología

Las actividades de aprendizaje que se consideran en esta práctica es la solución de los siguientes ejercicios:

1. Investigue cuales son la funciones polimórficas construidas en Haskell.
2. Investigue cuales son los tipos sobrecargados en Haskell.
3. Considere los resultados de las investigaciones previas para escribir un programa en Haskell que resuelva los siguientes problemas:
   a. cuadrados, que dada una lista de números, devuelva la lista de cuadrados.
   b. longitudes, que dada una lista de listas, devuelve la lista de sus longitudes.
   c. orden, que dada una lista de pares de números, devuelve la lista de aquellos pares en los que la primera componente es menor que el triple de la segunda.
   d. pares, que dada una lista de enteros, devuelve la lista de los elementos pares.
   e. letras, que dada una lista de caracteres, devuelve la lista de aquellos que son letras (minúsculas o mayúsculas)
   f. masDe, que dada una lista de listas xss y un numero n, devuelve la lista de aquellas listas de xss con longitud mayor que n.

10.7 Sugerencias didácticas

Analizar con los alumnos el concepto de polimorfismo y su implementación en otros lenguajes de programación y sus aplicaciones.

10.8 Evidencias de aprendizaje (Reporte del alumno)

Reporte de la práctica de acuerdo al Anexo A.

10.9 Bibliografía

Kurt Will, (2018), Get Programming with Haskell , Manning Publications.

Allen Christopher, Moronuki Julie, (2016), Haskell Programming: From First Principles, Gumroad.

Thomasson Samuli, (2016), Haskell High Perfomance Programming, Packt Publishing Limited.

Church James, Learning Haskell Data Analysis, Packt Publishing Limited.

Bird Richard, (2014), Thinking Functionally with Haskell, Cambridge University Press.

Programación Funcional

Apuntes y Prácticas

Práctica 11: Vectores en la programación funcional

11.1 Competencias especificas de la unidad.

Resolución de problemas que requieren del uso de vectores.

Diseñar programación con modularidad.

Solución de problemas.

Capacidad de aplicar los conocimientos en la práctica.

Habilidades de investigación.

Capacidad de aprender.

11.2 Objetivo de la Práctica

Conocer y aplicar los diferentes constructores para vectores en Haskell.

11.3 Introducción

El *arreglo* (array) es probablemente la estructura de datos mas conocida porque en casi todos los lenguajes de programación, es la única estructura disponible explícitamente. Un arreglo es una estructura homogénea: consiste de componentes los cuales son todos del mismo tipo. El arreglo es también llamada estructura de acceso aleatorio; todos los componentes pueden ser seleccionados aleatoriamente y son igualmente accesibles. Los arreglos pueden ser de una sola dimensión, también llamados vectores, los arreglos de dos dimensiones también llamados matrices o tablas.

Un vector $a \in F^n$, donde se tienen n elementos y cada uno de ellos es un miembro de un conjunto F. También se puede escribir a como $[a_n]$, lo cual es definido por sus elementos $a_i \in F, i = 1..n$.

En la programación, para denotar un vector se le da un nombre a la estructura, de tal manera que para denotar un componente individual del vector, el nombre de la estructura es usado junto con un nombre de *índice* (index) para seleccionar el componente. Este índice es un valor del tipo definido como el *tipo índice* del arreglo.

La definición de un arreglo tipo T por lo tanto especifica el tipo base T_0 y un índice tipo I.

$$tipo\ T = arreglo[I]\ de\ T_0$$

La definición y declaración de los vectores en los lenguajes de programación puede variar sintácticamente, pero su semántica es la misma.

Por ejemplo en el lenguaje C, la definición de un vector se muestra a continuación:

int valores[100];

donde se esta declarando un vector con nombre *valores* que va contener *100 números enteros (int)*.

En algunos otros lenguajes, los vectores se definen y se declaran bajo la semántica de alguna librería que este desarrollada para soportar este tipo de estructuras de datos. Haskell es un caso de este tipo de lenguajes. *Data.Vector* es una biblioteca de Haskell para trabajar con vectores. Esta biblioteca proporciona un buen rendimiento con una interfaz potente. Los tipos de datos principales que puede manejar son los *arrays boxed* y *unboxed*, que a su vez podrán ser *inmutables* o *mutables*. Además, los vectores pueden almacenar estructuras, son adecuados para la portabilidad desde y hacia C, y se indexan por valores enteros no negativos.

11.4 Correlación con el o los temas y subtemas del programa de estudio vigente

Tema: Evaluación perezosa.

Subtema: La estrategia de evaluación perezosa y técnicas de programación funcional perezosa.

11.5 Material y equipo necesario

Equipo de cómputo e Internet.

11.6 Metodología

Las actividades de aprendizaje que se consideran en esta práctica es la solución de los siguientes ejercicios:

1. Escriba un programa en Haskell que realice las siguientes funcionalidades:
   a. Crear un vector Datos con 10 números enteros.
   b. Calcular el promedio de los números almacenados en el vector Datos.
   c. Encontrar la posición del numero entero menor almacenado en el vector Datos.
   d. Encontrar la posición del numero entero mayor almacenado en el vector Datos.
2. Construya un programa en Haskell que genere N números aleatorios entre 0 y 100 y los almacene en un vector. Se debe de obtener la suma total de los números impares que pertenecen a ese rango.

3. Construya un programa en Haskell que lance un dado N veces y muestre cuantas veces fue lanzado cada lado del dado, en un vector se debe almacenar la cantidad de veces que se lanzo cada lado.

11.7 Sugerencias didácticas

Analizar con los alumnos el concepto de matemático del vector y asociarlo con su implementación en los lenguajes de programación.

11.8 Evidencias de aprendizaje (Reporte del alumno)

Reporte de la práctica de acuerdo al Anexo A.

11.9 Bibliografía

Kurt Will, (2018), Get Programming with Haskell , Manning Publications.

Allen Christopher, Moronuki Julie, (2016), Haskell Programming: From First Principles, Gumroad.

Thomasson Samuli, (2016), Haskell High Perfomance Programming, Packt Publishing Limited.

Church James, Learning Haskell Data Analysis, Packt Publishing Limited.

Bird Richard, (2014), Thinking Functionally with Haskell, Cambridge University Press.

Programación Funcional

Apuntes y Prácticas

Práctica 12: Matrices en la programación funcional

12.1 Competencias especificas de la unidad.

Resolución de problemas que requieren del uso de matrices.

Diseñar programación con modularidad.

Solución de problemas.

Capacidad de aplicar los conocimientos en la práctica.

Habilidades de investigación.

Capacidad de aprender.

12.2 Objetivo de la Práctica

Conocer y aplicar los diferentes constructores para matrices en Haskell.

12.3 Introducción

El *arreglo* (array) es probablemente la estructura de datos mas conocida porque en casi todos los lenguajes de programación, es la única estructura disponible explícitamente. Un arreglo es una estructura homogénea: consiste de componentes los cuales son todos del mismo tipo. El arreglo es también llamada estructura de acceso aleatorio; todos los componentes pueden ser seleccionados aleatoriamente y son igualmente accesibles. Los arreglos pueden ser de una sola dimensión, también llamados vectores, también se tienen los arreglos de dos dimensiones llamados matrices o tablas.

Una matriz $a \in F_{nxm}$, donde se tienen m filas y n columnas, donde cada elemento de la matriz denotado por $a_{i,j}$, es un miembro del conjunto F. También se puede escribir a como $[a_{m,n}]$, lo cual es definido por sus elementos $a_{i,j} \in F, i = 1..m, j = 1..n$.

En la programación, para denotar una matriz se le da un nombre a la estructura, de tal manera que para denotar un componente individual de la matriz, el nombre de la estructura es usado junto con un par de nombres de *índices* (index) para seleccionar el

componente. Este índice es un valor del tipo definido como el *tipo índice* del arreglo. La definición de un arreglo tipo *M* por lo tanto se especifica

$$M = arreglo[1..N]\ de\ Filas$$

o

$$M = arreglo[1..N]\ de\ arreglo[1..M]\ de\ tipo\ de\ dato$$

La definición y declaración de las matrices en los lenguajes de programación puede variar sintácticamente, pero su semántica es la misma.

Por ejemplo en el lenguaje C, la definición de una matriz se muestra a continuación:

int valores[100][100];

donde se esta declarando una matriz con nombre *valores* que va contener *10,000* (*100x100) números enteros (int).*

En algunos otros lenguajes, las matrices se definen y se declaran bajo la semántica de alguna librería que este desarrollada para soportar este tipo de estructuras de datos. Haskell es un caso de este tipo de lenguajes. En Haskell a las matrices también se les llama tablas. *Data.Array* es una biblioteca de Haskell para trabajar con tablas. Esta biblioteca proporciona un buen rendimiento con una interfaz potente.

12.4 Correlación con el o los temas y subtemas del programa de estudio vigente

Tema: Evaluación perezosa.

Subtema: La estrategia de evaluación perezosa y técnicas de programación funcional perezosa.

12.5 Material y equipo necesario

Equipo de cómputo e Internet.

12.6 Metodología

Las actividades de aprendizaje que se consideran en esta práctica es la solución de los siguientes ejercicios:

1. Escriba un programa en Haskell que realice las siguientes funcionalidades:
   a. Crear una tabla Datos con 100 números enteros, 10 columnas y 10 filas.
   b. Calcular el promedio de los números almacenados en la tabla Datos.
   c. Encontrar la posición del numero entero menor almacenado en la tabla Datos.
   d. Encontrar la posición del numero entero mayor almacenado en la tabla Datos.

2. Escriba un programa en Haskell que realice las siguientes funcionalidades:
   a. Crear una tabla Datos1 con 100 números enteros a partir de una lista de filas, considere la tabla con 10 columnas y 10 filas.

b. Obtener el numero de filas de la tabla Datos1.
c. Obtener el numero de columnas de la tabla Datos1.
d. Obtener la dimensión de la tabla Datos1.
e. Crear la tabla Datos2 con 100 números enteros a partir de una lista de filas, considere la tabla con 10 columnas y 10 filas.
f. Realizar la suma de las tablas Datos1 y Datos2, almacenar el resultado en la tabla Datos3.
g. Realizar la multiplicación de las tablas Datos1 y Datos2, almacenar el resultado en la tabla Datos4.
h. Convertir el resultado de la tabla Datos3 en una lista.
i. Convertir el resultado de la tabla Datos4 en una lista.

12.7 Sugerencias didácticas

Analizar con los alumnos el concepto de matemático de la matriz y asociarlo con su implementación en los lenguajes de programación.

12.8 Evidencias de aprendizaje (Reporte del alumno)

Reporte de la práctica de acuerdo al Anexo A.

12.9 Bibliografía

Kurt Will, (2018), Get Programming with Haskell , Manning Publications.

Allen Christopher, Moronuki Julie, (2016), Haskell Programming: From First Principles, Gumroad.

Thomasson Samuli, (2016), Haskell High Perfomance Programming, Packt Publishing Limited.

Church James, Learning Haskell Data Analysis, Packt Publishing Limited.

Bird Richard, (2014), Thinking Functionally with Haskell, Cambridge University Press.

Práctica 13: Bolsas en la programación funcional

13.1 Competencias especificas de la unidad.

Resolución de problemas que requieren del uso de bolsas.

Diseñar programación con modularidad.

Solución de problemas.

Capacidad de aplicar los conocimientos en la práctica.

Habilidades de investigación.

Capacidad de aprender.

13.2 Objetivo de la Práctica

Conocer y aplicar los diferentes constructores para bolsas en Haskell.

13.3 Introducción

En las estructuras de datos pila y cola, el orden en que los elementos se colocan en el contenedor es importante, porque los elementos se relacionan con el orden en que están insertados. Para la bolsa (bag), el orden de inserción es completamente irrelevante. Los elementos se pueden insertar y eliminar de manera aleatoria. Al usar el nombre Bolsa para describir este tipo de datos abstractos, la intención es para sugerir una vez más ejemplos de colección que le resultarán familiares el usuario de su experiencia cotidiana. Una bolsa de canicas es una buena imagen mental. Las operaciones que se pueden hacer con una bolsa incluyen insertar una nuevo valor, eliminar un valor, probar para ver si un valor se mantiene en la recopilación y determinar el número de elementos en la colección.

Un conjunto (set) extiende la bolsa de dos maneras importantes. Primero, los elementos en un conjunto deben ser únicos; agregar un elemento a un conjunto cuando ya está contenido en la colección no tendrá efecto. En segundo lugar, el conjunto agrega una serie de operaciones que combinan dos conjuntos y se produce un nuevo conjunto. Algunas implementaciones de un conjunto permiten que los elementos se repitan mas de una vez. Esto generalmente se denomina *multiset*.

La bolsa es la más básica de las estructuras de datos de colección, y por lo tanto, casi cualquier aplicación no requiere recordar el orden en que se insertan los elementos. Tome, por ejemplo, un corrector ortográfico. Un verificador en línea colocaría un diccionario de palabras correctamente deletreadas en una bolsa. Cada palabra en el archivo es luego probada contra las palabras en la bolsa, y si no se encuentra, está marcado. Un verificador fuera de línea podría usar establecer operaciones. Las palabras correctamente deletreadas se pueden colocar en una sola bolsa, las palabras en el documento colocado en un segundo y la diferencia entre los dos calculados. Palabras encontradas en el documento, pero no el diccionario podría ser impreso.

13.4 Correlación con el o los temas y subtemas del programa de estudio vigente

Tema: Evaluación perezosa.

Subtema: La estrategia de evaluación perezosa y técnicas de programación funcional perezosa.

13.5 Material y equipo necesario

Equipo de cómputo e Internet.

13.6 Metodología

Las actividades de aprendizaje que se consideran en esta práctica es la solución de los siguientes ejercicios:

Un bag (o multiset) es una colección de elementos. Cada elemento puede ocurrir un o más veces en el bag. Elija una representación eficiente para bags. Sería deseable que cada bag tenga una representación única. Definir las siguientes operaciones sobre bags, asegurándose de que aquellas que devuelvan bags lo hagan mediante una representación válida.

a) list2bag, que toma una lista de elementos y devuelve un bag que contiene esos elementos. El número de ocurrencia de un elemento en la lista y en el bag resultante debe ser el mismo.

b) bagEmpty, que devuelve True exactamente cuando el bag dado está vacío.

c) bagCar, que devuelve su cardinalidad (cantidad total de ocurrencias de todos los elementos).

d) bagElem, que devuelve verdadero si un elemento dado está en el bag dado;

e) bagOccur, que devuelve True si y sólo si dos bags tienen los mismos elementos con la misma cantidad de ocurrencias.

f) bagEqual, que devuelve True si y sólo si dos bags tienen los mismos elementos con la misma cantidad de ocurrencias.

g) bagSubbag, que devuelve True si y sólo si el primer bag es subbag del segundo; se dice que X es subbag de Y si cada elemento de X ocurre en Y al menos tantas veces como en X .

h) bagInter, que toma dos bags y calcula el bag intersección. La intersección de dos bags X e Y contiene a todos los elementos comunes a X e Y con la menos cantidad posible de ocurrencias.

i) bagSum, que calcula el bag unión de dos bags dados; el bag suma de los dos bags X e Y consiste de todos los elementos de X e Y con número de ocurrencias de suma de las ocurrencias en cada uno de estos bags.

j) bagInsert, que toma un elemento y un bag y devuelve el bag con el elemento insertado.

k) bagDelete, que toma un elemento y un bag, y devuelve el bag con el elemento borrado.

13.7 Sugerencias didácticas

Analizar con los alumnos el concepto de matemático de conjuntos y asociarlo con su implementación en los lenguajes de programación.

13.8 Evidencias de aprendizaje (Reporte del alumno)

Reporte de la práctica de acuerdo al Anexo A.

13.9 Bibliografía

Kurt Will, (2018), Get Programming with Haskell , Manning Publications.

Allen Christopher, Moronuki Julie, (2016), Haskell Programming: From First Principles, Gumroad.

Thomasson Samuli, (2016), Haskell High Perfomance Programming, Packt Publishing Limited.

Church James, Learning Haskell Data Analysis, Packt Publishing Limited.

Bird Richard, (2014), Thinking Functionally with Haskell, Cambridge University Press.

Programación Funcional

Apuntes y Prácticas

Práctica 14: Entrada y salida en la programación funcional

14.1 Competencias especificas de la unidad.

Resolución de problemas que requieren el uso de entrada y salida de datos.

Diseñar programación con modularidad.

Solución de problemas.

Capacidad de aplicar los conocimientos en la práctica.

Habilidades de investigación.

Capacidad de aprender.

14.2 Objetivo de la Práctica

Conocer y aplicar los diferentes constructores para la entrada y salida de datos en Haskell.

14.3 Introducción

El mecanismo más simple de entrada y salida de datos es leer de la entrada estándar y escribir en la salida estándar, las cuales se encuentran en la terminal del usuario (teclado y pantalla respectivamente). Esto es útil cuando la cantidad de datos es pequeña. Sin embargo, cuando la cantidad de datos es grande es necesario captarlos, almacenarlos, procesarlos y visualizarlos con las técnicas correspondientes a cada operación.

Las estructuras de datos aplicadas a colecciones de datos en almacenamiento secundario son llamadas organizaciones de archivo. Los datos pueden ser representados, organizados y almacenados de la forma más eficiente posible, lo cual dependerá obviamente de la aplicación y de la selección de la estructura de datos más adecuada, misma que se transformará según la complejidad que se requiera.

El concepto de archivo en el contexto computacional pareciera muy simple, ya que si partimos de su definición encontrada en diversos textos, que nos dice que “un archivo es una secuencia de bytes”, entonces cómo es posible que podamos realizar el procesamiento de datos, escribir textos, escuchar música, ver fotografías, diseñar

piezas mecánicas, etc., donde en todas estas actividades, ya muy cotidianas, hemos usado archivos y sin embargo, sólo tenemos una secuencia de bytes.

Nuestra respuesta al anterior argumento esta basada también en conceptos sencillos, como lo son la organización, el formato y la abstracción de los datos. Lo cual nos permite crear diversos tipos de archivos para su posterior uso en las aplicaciones computacionales.

En el entorno computacional, el concepto de archivo lo podemos encontrar bajo diferentes contextos, uno de ellos es el sistema operativo, que tiene como uno de sus objetivos la administración de los archivos almacenados en los dispositivos de almacenamiento secundario. La estructura usada esta basada en archivos.

Otro contexto es el procesamiento de datos, en el cual se tienen diversas organizaciones que han permitido un eficiente almacenamiento y recuperación de los datos. Esto mismo se puede extender al contexto de las bases de datos, las cuales usan las mismas organizaciones de archivos que las que podemos usar en un sistema informático, pero incrementando la abstracción y haciéndolo mas simple para el usuario. De tal manera que encontraremos diversos tipos de archivos, con organizaciones y formatos diferentes.

Primeramente se define el archivo de datos. Los datos pueden ser representados por números y caracteres, de tal modo que un dato como el nombre de un empleado, el número de matrícula de un alumno o un precio, pueden ser identificados por una entidad a la cual se le llama Campo, entonces obtendríamos los campos Nombre, Matricula y Precio como se muestra en la tabla 1.1.

Tabla 1.1 Campos

Nombre	*Matricula*	*Precio*
Alma Patricia	*74060829*	*30000.00*

En este caso los campos no tienen ninguna relación lógica, ya que se trata de diversos contextos; pero, ¿qué pasa si para cierta aplicación específica como una nómina, se consideran los campos mostrados en la tabla 1.2?

Tabla 1.2 Interrelación lógica entre campos

Nombre	*Numero Empleado*	*Sueldo*
Alma Patricia	*12345*	*23456.00*

Aquí ya existe una relación lógica y encontramos una nueva entidad llamada Registro, la cual puede ser definida como:

Una colección ordenada y finita de elementos posiblemente heterogéneos que son tratados como unidad.

Es necesario diferenciar entre un arreglo y un registro. El arreglo contiene el mismo tipo de datos y el registro puede contener diferentes tipos de datos. Conceptualmente, un conjunto de arreglos en paralelo sería equivalente a varios registros. Como se puede apreciar en la figura 8, un registro esta compuesto de uno o más campos, estos posiblemente de datos heterogéneos.

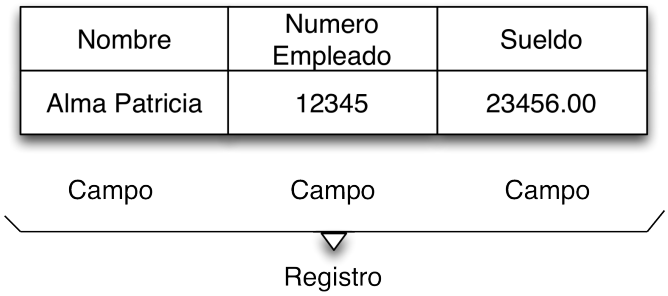


Figura 8. Estructura de un registro

Por lo regular, los registros contienen un campo de identificación el cual normalmente recibe el nombre de llave o clave primaria, en el caso anterior puede ser el número de empleado. Dado que un archivo está formado por un conjunto de registros, este campo de identificación nos permite individualizar los registros de tal manera que permite su plena identificación.

La clave de identificación o llave generalmente es un código sencillo y práctico, con las características siguientes: no deberá ser ambiguo, debe ser preciso y permitir distinguir un caso de otro, es decir, debe particularizar un caso con respecto a los demás. Esta clave de identificación es usada para llevar a cabo la clasificación de los datos y para el acceso a los datos asociados a este valor de llave o clave.

Ya mencionamos que en un registro existe una relación lógica de los campos, esto es, los atributos relacionados de una aplicación: Nombre, Sueldo, Dirección, etc. Si se tiene una colección de registros de la aplicación Nómina relacionados de manera lógica (todos son empleados, no se trata de registros de Inventario o de Contabilidad) y son tratados como una unidad se le llama Archivo; la estructura lógica entre las unidades definidas anteriormente se ilustra en la figura 9.

Figura 9. Estructura lógica

En conclusión, un conjunto de campos relacionados lógicamente constituyen un registro, y un conjunto de registros relacionados lógicamente constituye un archivo.

Existen por lo menos tres buenas razones para estructurar una colección de datos como un archivo, obviamente éstas son diferentes del motivo principal para el uso de un archivo, que es la persistencia de los datos:

- Una colección de datos puede ser almacenada como un archivo porque puede ser tan grande para poder mantenerla en memoria principal.
- Puede ser almacenada como un archivo porque únicamente una pequeña porción de la colección es accedida por un programa en cualquier tiempo, haciendo incongruente almacenar la colección entera en memoria principal.
- Si la colección de datos es muy pequeña, puede ser deseable retener la colección de datos independiente de la ejecución de cualquier programa particular.

14.4 Correlación con el o los temas y subtemas del programa de estudio vigente

Tema: Evaluación perezosa.

Subtema: La estrategia de evaluación perezosa y técnicas de programación funcional perezosa.

14.5 Material y equipo necesario

Equipo de cómputo e Internet.

14.6 Metodología

Las actividades de aprendizaje que se consideran en esta práctica es la solución de los siguientes ejercicios:

1. Construya un programa en Haskell que simula una máquina que vende golosinas. La máquina solo vende 3 productos:

   Chocolates por $8.00

   Papitas por $6.00

   Goma de mascar por $4.00

   Las entradas son las siguientes:

   a. El usuario pueda introducir la cantidad de dinero.
   b. Seleccionar cuantos productos de cada uno se quiere comprar.

   El programa debe verificar si la cantidad de dinero es mayor o igual al costo total de la compra seleccionada. Si no hay suficiente dinero se debe mostrar un mensaje que indique "Compra Inválida" y muestra la cantidad de dinero que hace falta. En caso contrario se muestra el mensaje "Compra Válida".

2. Escriba un programa en Haskell que imprima el contenido de un archivo de texto, indicando el número de línea y el texto correspondiente de la línea.

3. El formato de un archivo CSV es muy sencillo, sin embargo, su uso se puede encontrar en muy diversas aplicaciones, por ejemplo en teléfonos móviles, en PLC (controladores lógicos programables), en hojas electrónicas, etc. Se puede considerar como un pseudo-estándar para el intercambio de datos.

   Asuma que se tiene un archivo con este formato y como se puede apreciar en las siguientes líneas, éstas contienen campos los cuales están separados por comas.

   Hora, Cuarto, Paciente, Medico

   08:00,115,JOSE RUIZ,PEREZ PEREZ

   08:30,112,JUAN PEREZ,AVILA AVILA

   09:00,200,ALICIA GOMEZ,ZAZUETA ZAZUETA

   09:30,210,ANDREA PALMA,ZAZUETA ZAZUETA

   09:40,300,LUIS DE LA VARA,PEREZ PEREZ

   09:45,218,ROBERTO CALIDA,AVILA AVILA

   10:00,315,JOSE RODRIGUEZ,ZAZUETA ZAZUETA

   Escriba un programa en Haskell que permita leer un archivo como el anterior y crear una estructura de datos en memoria para mantener esta información, una vez creada la estructura de datos el programa debe permitir realizar las siguientes consultas:

   ¿Quienes son los pacientes del Médico "X"?

   ¿A que hora ingreso el paciente "X"?

   ¿En cual cuarto esta el paciente "X"?

14.7 Sugerencias didácticas

Los alumnos deben realizar ejercicios de entrada y salida estándar a nivel del sistema operativo. Asimismo, usando comandos del sistema operativo leer archivos planos de texto.

14.8 Evidencias de aprendizaje (Reporte del alumno)

Reporte de la práctica de acuerdo al Anexo A.

14.9 Bibliografía

Kurt Will, (2018), Get Programming with Haskell , Manning Publications.

Allen Christopher, Moronuki Julie, (2016), Haskell Programming: From First Principles, Gumroad.

Thomasson Samuli, (2016), Haskell High Perfomance Programming, Packt Publishing Limited.

Church James, Learning Haskell Data Analysis, Packt Publishing Limited.

Bird Richard, (2014), Thinking Functionally with Haskell, Cambridge University Press.

Práctica 15: Lógica de primer orden

15.1 Competencias a desarrollar

Comprender los elementos de la lógica de predicados.

Capacidad de análisis y síntesis.

Resolución de problemas.

Habilidad para buscar y analizar información proveniente de fuentes diversas.

Habilidades de investigación.

Capacidad de aprender.

15.2 Objetivo de la Práctica

Construir sentencias en lógica de predicados.

15.3 Introducción

El lenguaje, como instrumento de comunicación del conocimiento humano, está constituido por frases de tipo interrogativo, de tipo imperativo y de tipo declarativo. Estas últimas constituyen el elemento básico de la descripción del conocimiento.

El conocimiento puede producirse bien por constataciones de hechos o ideas, que tienen su reflejo en frases de tipo declarativo, como por deducción, a partir de una serie de declaraciones, de otras nuevas cuya afirmación se obtiene de las declaraciones previas. Esa producción de conocimiento es posible a través de formulaciones matemáticas de la lógica, más usuales en su etapa previa de simbolización de las formas de lenguaje natural que puede hacerse a dos niveles según el grado de complejidad del análisis: Lógica Proposicional y Lógica de Predicados.

- *Lógica Proposicional*: Representación del lenguaje natural tomando como elemento básico de la formulación una representación matemática de las frases declarativas simples (proposiciones).
- *Lógica de Predicados*: Representación del lenguaje natural tomando como base los componentes de algunos tipos de proposición; Términos y Predicados.

15.3.1 Lógica Proposicional

El desarrollo del cálculo proposicional se basa en entidades matemáticas representativas de unidades de información, cuya estructura se contempla como un todo, sin diferenciar sus componentes.

Este planteamiento no permite representar matemáticamente determinadas estructuras deductivas, que, sin embargo, son correctas en el lenguaje natural.

Así, por ejemplo, el razonamiento:

"Si en la luna hay agua, entonces en la luna hay vida",

"No ocurre que en la luna hay vida",

"Luego, no es cierto que en la luna hay agua"

Esta estructura deductiva, tratada con la hipótesis que sirve de base al cálculo proposicional y considerando las siguientes definiciones:

$$p = en\ la\ luna\ hay\ agua$$

$$q = en\ la\ luna\ hay\ vida$$

tendría la siguiente representación matemática:

$$p \rightarrow q$$

$$\neg q$$

$$\therefore \neg p$$

Por tanto, ninguna de las proposiciones p y q pueden describirse mediante partes de las mismas dotadas de significado propio, unidas por conectivos, que sean comunes en algunas de ellas y, por tanto, la relación entre premisas y conclusión que hace la deducción correcta, no puede detectarse con este nivel de representación.

Otro aspecto a considerar es que a partir de las oraciones gramaticalmente posibles en las lenguas naturales, debemos de saber que no todas pueden ser simbolizadas, sino sólo aquellas que o son falsas o verdaderas, es decir las oraciones declarativas ya sean simples o compuestas. Por otro lado al recurrir al lenguaje natural como un medio para realizar argumentaciones, tenemos que considerar la estructura lingüística, el significado léxico y los principios del uso del lenguaje para hacer formalizaciones adecuadas.

El proceso de traducción del lenguaje natural al lenguaje formal de la lógica es complejo y requiere de mucha práctica, ya que no existen reglas fijas. Dado que los estados iniciales del diseño de un sistema computacional (ya sea una aplicación de gestión de la información o un sistema inteligente) las informaciones sobre el sistema se representan muchas veces en lenguaje natural no formalizado, el dominio de las técnicas de traducción es de vital importancia para el informático.

Aunque como se ha dicho, no hay reglas precisas para resolver este problema, pero podemos seguir ciertas pautas sencillas:

1. Primero, identificar los enunciados simples y, después, las conectivas (utilizando el significado que se le ha atribuido a los conectores).
2. Finalmente, asignar una letra enunciativa a cada uno de los enunciados identificados y aplicar las normas de escritura abreviada de formas enunciativas.

A continuación se muestran un ejemplo de lo anterior:

Enunciado:

"Si llueve se terminarán los problemas de sequía y no hará falta más dinero"

Análisis de los enunciados simples y asignación de letras enunciativas:

llueve = p

se terminarán los problemas de sequía = q

hará falta más dinero = r

Formalización: $p \rightarrow (q \ \wedge \neg r)$

Es importante mencionar que para realizar el cálculo proposicional, esto es, llevar a cabo el proceso de interpretación, en ocasiones se deben usar equivalencias, éstas, están establecidas en axiomas, aunque es importante aclarar que no es de interés de este trabajo realizar la interpretación con lógica proposicional, sin embargo estos axiomas son usados también en la lógica de predicados.

15.3.2 Lógica de Predicados

Para tratar matemáticamente las estructuras deductivas es preciso crear una teoría que no tome como base la simbolización matemática de la proposición total sino la de sus componentes, es decir :

- Qué se afirma.

- De quién o quiénes se afirma.

El primer elemento se define como el predicado y el segundo, como los sujetos o términos. Así, en la frase:

- *Juan es hombre*,

<*es hombre*> es el predicado y <*Juan*> el sujeto o término de la proposición.

Puede haber proposiciones con varios términos. Los predicados que se refieren a un único término se denominan predicados absolutos o monádicos. Los que se refieren a varios sujetos se denominan predicados de relación o políadicos.

Una vez definidos los componentes de la proposición se plantea su representación matemática en base a términos y predicados. El análisis y simbolización matemática

de los elementos integrantes de una proposición requiere la definición de una sintaxis para la construcción de <formas> (fórmulas) representativas de estructuras de frases del lenguaje usual, acorde con las convenciones de este último y las exigencias de transparencia, economía y precisión de la matemática.

La definición del lenguaje formal para construcción de fórmulas que se describe a continuación es una generalización del lenguaje para cálculo proposicional.

• Alfabeto

Consta de los siguientes símbolos :

a) Símbolos de términos constituidos por letras de variables (últimas letras del alfabeto; *x, y, z, t, w*, etc.) y letras de constantes (primeras letras del alfabeto; *a, b, c, d*, etc.).

b) Símbolos de predicado, se emplearán las letras minúsculas del alfabeto; *p, q, r, s,* etc.

c) Símbolo de conectivos y paréntesis idénticos que en el cálculo proposicional: $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, (,).

d) Símbolos de cuantificación:

Cuantificador universal: $\forall$

Cuantificador existencial: $\exists$

• Sintaxis de construcción de Fórmulas

Una fórmula constituye una sucesión de símbolos del alfabeto que verifica las siguientes reglas de formación:

e) Toda proposición es una fórmula.

f) Si *p* es una letra de predicados de *n* plazas, *p(t1, t2,...tn)* es una fórmula, siendo *ti* el símbolo de términos.

g) Si *A* es una fórmula que contiene libre la variable *Xi*:

$\forall A\ (X1, X2,...Xi,...,Xn)$

$\exists A\ (X1, X2,...Xi,...,Xn)$

Son fórmulas correctas sintácticamente; las otras variables *Xk* que figuran en *A* distintas de *Xi* continúan libres.

h) Si *A* y *B* son fórmulas.

$\neg A,\ A \vee B,\ A \wedge B,\ A \rightarrow B$ son fórmulas.

i) Sólo son fórmulas aquellas construidas según los conceptos de a) hasta h).

El uso de variables y de los símbolos para la cuantificación hace que la lógica de predicados sea mas poderosa que la lógica proposicional, como se mencionó anteriormente los cuantificadores son:

∀ Cuantificador universal (para todos, todos, cada uno, cualquiera, etc.)

∃ Cuantificador existencial (existe, algunos, al menos uno, etc.)

De tal manera que el rango o alcance de un cuantificador corresponde a lo largo de un paréntesis que aparece inmediatamente después del cuantificador, para la sentencia *inteligente(alicia)* se puede apreciar que el termino individual es una constante, por lo que no se requiere de un cuantificador.

A nivel de relación entre conjuntos, en el caso de la cuantificación universal se da una relación de contención y en el caso de la cuantificación existencial, una relación de intersección. Por lo tanto existe una relación entre cuantificadores y conectivos función de verdad:

$$\forall\, y \;\rightarrow$$

$$\exists\, y \;\wedge$$

Cuando la importancia se traslada al *Universo del Discurso*, la cuantificación es restringida. Llamaremos de esta forma al conjunto al cual pertenecen los valores que puedan tomar las variables. Lo notaremos por *U* y lo nombraremos por conjunto universal o, simplemente, universo. Debe contener, al menos, un elemento.

Ejemplo. En una posible evaluación del predicado *"p(x) : x > 5"*, elegiríamos probablemente un conjunto numérico, por ejemplo los números enteros, como universo del discurso. No tendría sentido elegir el conjunto de los colores del arco iris ya que podríamos encontrarnos con situaciones tales como *"azul > 5"*.

También es importante considerar el orden de los cuantificadores:

Cada uno tiene alguien a quien admira. $\forall x\, \exists y$ *Admira(x,y)*

Hay alguien a quien cada uno admira. $\exists x\, \forall y$ *Admira(x,y)*

Asimismo la negación es importante, como se puede ver en los siguientes ejemplos:

Todos no vinieron a clase. $\forall x\, \neg$*Vinieron(x,clase)*

No todos vinieron a clase. $\neg\forall x$ *Vinieron(x,clase)*

Variantes del cuantificador Existencial

Cuantificador existencial para unicidad *(∃! o ∃=1)*:

Para un enunciado abierto *P(x)*, la proposición *∃!x P(x)*, se lee "existe un x único tal que *P(x)",* es verdad si y sólo si el conjunto verdad de *P(x)* tiene exactamente un elemento. El símbolo ∃! se llama cuantificador existencial de unicidad.

Ejemplo 1: para los números naturales.

(∃!x) (x ≥ 1) es la proposición: existe un único *x* tal que *x ≥ 1* y es falso.

(∃!x) (x = 1/2) es la proposición: existe un único *x* tal que *x = 1/2* y es falso.

(∃!x) (x ≤ 1) es la proposición: existe un único *x* tal que *x ≤ 1* y es verdad.

Ejemplo 2: Únicamente un estudiante fallo en Historia.

(∃!x) (estudiante(x) ∧ fallo(x,Historia)) lo cual es equivalente a:

(∃x) [(estudiante(x) ∧ fallo(x,Historia)) ∧ (∀y (estudiante(y) ∧ fallo(y,Historia))) → x=y]

Cuantificador existencial para definir limites

Juan tiene al menos tantos perros como tiene Maria.

Esto es imposible expresarlo con lo tradicional *∃x,y,z*, entonces se puede modelar así:

∃n [$\exists_n^n$ *x perro(x) ∧ dueño(maria,x) ∧* $\exists_{n+1}$*y perro(y) ∧ dueño(juan,y)]*

lo cual complica las pruebas o para realizar el razonamiento.

15.4 Correlación con el o los temas y subtemas del programa de estudio vigente

Tema: Fundamentos de programación lógica.

Subtema: Repaso de la lógica de primer orden.

15.5 Material y equipo necesario

Equipo de cómputo e Internet.

15.6 Metodología

Las actividades de aprendizaje que se consideran en esta práctica son las siguientes:

1. Usando lógica de predicados obtenga las fórmulas bien formadas (wwfs) de las siguientes sentencias:
   a. Todos los leones son fieras.
   b. Algunos leones no toman café.
   c. Algunas criaturas fieras no toman café.
   d. Todos los gatos son blancos.
   e. Algunos gatos no son blancos.
   f. Algunos gatos son blancos.
   g. Todos los gatos no son blancos.
   h. Ningún gato es blanco.

2. Usando lógica de predicados obtenga las wwfs de los siguientes ejercicios:
   2.1Considere los siguientes axiomas:

   a) Todos los coyotes cazan algún correcamino.

   b) Cada correcamino que dice “beep-beep” es inteligente.

   c) Ningún coyote atrapa algún correcamino.

d) Cualquier coyote que caza algún correcamino pero no lo atrapa esta frustado.

2.2 Considere los siguientes axiomas:

a) Cualquiera que maneje una Harley es de rudo carácter.

b) Cada motociclista maneja algo que es una Harley o una BMW.

c) Cualquiera que maneje cualquier BMW es un yuppie.

d) Cada yuppie es un abogado.

e) Cualquier niña bonita no sale con alguien que sea de rudo carácter.

f) María es una niña bonita y Juan es un motociclista.

2.3 Considere los siguientes axiomas:

a) Cada inversionista compra algo que esta en la bolsa o en bonos.

b) Si el Promedio Down-Jones cae, entonces todas las bolsas que no están en oro también caen.

c) Si la tasa de interés T-Bill sube, entonces todos los bonos caen.

d) Cada inversionista que compró algo que cae no es feliz.

2.4 Considere los siguientes axiomas:

a) Todos los niños aman a todos los dulces.

b) Cualquiera que ama cualquier dulce no es un fanático de la nutrición.

c) Cualquiera que come cualquier calabaza es un fanático de la nutrición.

d) Cualquiera que compra cualquier calabaza la corta o la come.

e) Juan compra una calabaza.

f) La panocha es un dulce.

15.7 Sugerencias didácticas

Realizar ejercicios de lógica de predicados.

15.8 Evidencias de aprendizaje (Reporte del alumno)

Reporte de la práctica de acuerdo al Anexo A.

15.9 Bibliografía

P. J. Irnazo, (2005), Lógica Simbólica para Informáticos, Alfaomega – Rama.

C.L. Chang & R.C. Lee, (1973), Symbolic Logic and Mechanical Theorem Proving, Academic Press.

E. Suples, (1988), Lógica Matemática, Reverté.

Práctica 16: Unificación y resolución

16.1 Competencias a desarrollar

Comprender el proceso de deducción de la programación lógica.

Comprender los elementos de la lógica de predicados.

Resolución de problemas.

Capacidad de análisis y síntesis.

Habilidad para buscar y analizar información proveniente de fuentes diversas.

Habilidades de investigación.

Capacidad de aprender.

16.2 Objetivo de la Práctica

Realizar procesos deductivos con el principio de resolución.

16.3 Introducción

Una vez formalizada la estructura de las frases, es preciso representar matemáticamente las estructuras deductivas. Una estructura deductiva es una representación formal de un proceso de razonamiento para obtener una conclusión a partir de unas premisas.

Estas estructuras deductivas correctas se definen como elementos de un sistema construido mediante un conjunto de axiomas, unas reglas de inferencia y una definición de deducción, donde encontramos los axiomas de Kleene, las reglas de inferencias modus ponens, la generalización universal condicional y la generalización existencial condicional, así como los teoremas de la deducción del sistema de Kleene.

La teoría semántica del cálculo de predicados se construye no en base a la inscripción de las fórmulas o estructuras deductivas en un determinado armazón formal, sino en base a la atribución de significado de las fórmulas. Por tanto, forman parte del sistema aquellas fórmulas o estructuras deductivas cuyo significado cumple determinadas condiciones.

Al ser las fórmulas del cálculo de predicados más complejas que las del cálculo proposicional, la interpretación de las mismas requiere tener en cuenta mayor número de elementos.

16.3.1 Interpretación en lógica de predicados

Una interpretación es un mecanismo que permitirá asignar un valor Verdadero (T) o Falso (F) a una fórmula. Para ello es necesario dar un significado a cada uno de los símbolos que aparecen en la fórmula. Las conectivas y los cuantificadores tienen un significado fijo, mientras que el significado de las constantes, símbolos de función y símbolos de predicado puede variar.

Una interpretación de una fórmula *G* en lógica de predicados consiste de un dominio *D* no vacío y una asignación de valores a constantes, funciones y predicados.

1. A cada constante asignar un elemento en *D*.

2. A cada función asignar $D \times D \times .. \times D \rightarrow D$.

3. A cada predicado asignar *T*/*F*.

Por ejemplo, sea $\forall x\ P(x)$ y $\exists x\ \neg P(x)$

Una interpretación es la siguiente:

Dominio $D = \{1,2\}$

Asignación para *P*: *P(1) es T y P(2) es F*

$\forall x\ P(x)$ es *F* en esta interpretación porque *P(x)* no es *T* para x=1 y x=2.

Por otro lado, ya que $\neg P(2) = T$ en esta interpretación, $\exists x\ \neg P(x)$ es *T* en esta interpretación.

Una vez que se desarrolló la base teórica formalizada de la lógica de primer orden se llega al desarrollo de una serie de procedimientos para decisión de validez de fórmulas, estos procedimientos se le denominan métodos de demostración automática de teoremas.

Definiciones

Si *G* es una fórmula:

G es consistente si y sólo si existe una interpretación *I* tal que *G* evalúa a

verdadero en *I*, el cual es llamado modelo de *G*.

G es inconsistente si y sólo si es falso bajo todas las interpretaciones de todos los dominios.

G es válida si y sólo si *G* es verdad bajo todas las interpretaciones de todos los dominios.

G es una consecuencia lógica de *F1,F2,..,Fn* si y sólo si para cada interpretación *I*, si para $F1 \wedge F2 \wedge ... \wedge Fn$ es verdad en *I* entonces *G* es verdad en *I*.

Para mostrar consecuencia lógica:

Si $F1 \wedge F2 \wedge ... \wedge Fn \rightarrow G$ es válida, se prueba por Tabla Verdad, CNF (Forma Normal Conjuntiva) ó DNF (Forma Normal Disyuntiva).

ó

$F1 \wedge F2 \wedge ... \wedge Fn \ \wedge \ \neg G$ es inconsistente, prueba por resolución.

Existen diversos métodos para la demostración automática, nuestra atención se centrará en los métodos basados en el principio de resolución de Robinson que constituye junto con el algoritmo de unificación la base para la formulación de intérpretes de sistemas de programación lógica.

16.3.2 Procedimiento para obtener las cláusulas

En la figura 10 se muestran los pasos del procedimiento para obtener un modelo lógico que permita llevarlo posteriormente hacia la implementación computacional.

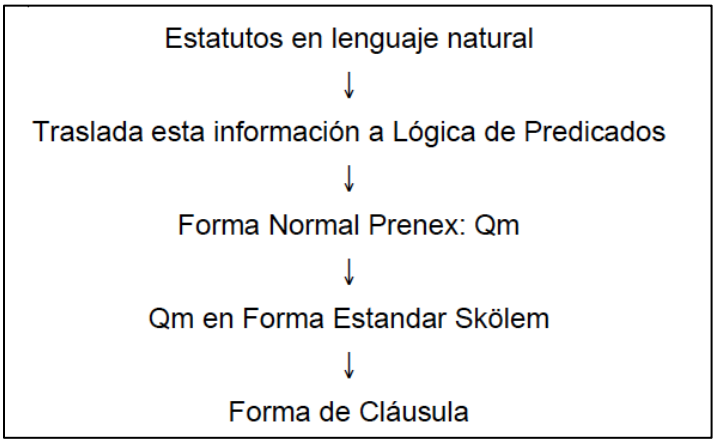


Figura 10. Del lenguaje natural a cláusulas

Los dos primeros pasos ya fueron descritos previamente y ahora se revisarán las formas Prenex, Skölem y clausular.

Forma Normal Prenex

Se dice que una fórmula *F* está en Forma Normal Prenex (FNP) si tiene la forma:

$$(Q1x1)(Q2x2)....(QnXn)\ (M)$$

donde $Q1,Q2,\ldots,Qn$ son cuantificadores (∀,∃) y M es una fórmula sin cuantificadores.

La secuencia *(Q1x1)(Q2x2)....(QnXn)* se denomina prefijo de *F*, mientras que *M* será la matriz de la fórmula.

Los pasos para convertir una fórmula a FNP son los siguientes:

1. Remover →, ↔, ¬¬
2. Use las leyes de Morgan para desplazar ¬ hacia adentro.
3. Use las leyes para mover cuantificadores a la izquierda.

Forma Estandar Skölem (SSF)

Una formula está en forma normal de Skölem si está en forma normal Prenex y todos los cuantificadores son universales. Para convertir una fórmula a forma normal Skölem es necesario suprimir todos los cuantificadores existenciales. La fórmula obtenida no será lógicamente equivalente a la anterior, sino será equisatisfacible.

A continuación se muestra el algoritmo de skolemización para suprimir los cuantificadores existenciales de una fórmula:

Sea *F = (Q1x1)(Q2x2)... (Qrxr) ...(Qnxn) (M)*, asuma que *M* está en FNC y *Qr* = ∃

1. Si no aparecen cuantificadores universal antes de *Qr*, entonces reemplace *xr* por una constante *c* que no esté en *M* y borre *Qrxr*. Por ejemplo:

   $\exists x \, \forall y \, P(x,y) = \forall y \, P(a,y)$

2. Si *Qa1x1….Qamxm* son cuantificadores universales antes de *Qr donde 1<= a1 <= am < r*, entonces elija un símbolo función *f* en el lugar *m*, reemplace *xr* por *f(x1,….,xm)* en *M* y borre *Qrxr.*
3. Aplique 1 y 2 para todo cuantificador ∃. Por ejemplo:

   $\forall x \, \forall y \, \exists z \, \forall w \, P(x,y,z,w)$ es equisatisfacible a

   $\forall x \, \forall y \, \forall w \, P(x,y,f(x,y),w)$

Conversión a Forma Clausular

Una vez que la forma SSF es obtenida, remueva todos cuantificadores universales ∀, ya que todas las variables son entendidas a ser cuantificadas universalmente.

16.3.3 Principio de Resolución

Si la base de conocimiento *KB* es un conjunto de sentencias $\{\alpha_1, \alpha_2, .., \alpha_n\}$, hay varias maneras equivalentes de formular la tarea de razonamiento deductivo:

$$KB \models \alpha$$

$$iff \models [(\{\alpha_1 \wedge \alpha_2 \wedge .. \wedge \alpha_n) \rightarrow \alpha]$$

$$iff \; KB \cup \{\neg\alpha\} \; es \; no \; satisfacible$$

$$iff \; KB \cup \{\neg\alpha\} \models \neg TRUE$$

donde *TRUE* es cualquier sentencia válida como $\forall x \; (x=x)$. Lo que esto significa es que si tenemos un procedimiento para probar la validez de sentencias o para probar la satisfacción de sentencias o para determinar si o no $\neg TRUE$ es una consecuencia lógica. Entonces ese procedimiento puede ser también usado para encontrar las consecuencias lógicas de una KB finita. Esto es significante, porque el procedimiento de Resolución es de hecho un procedimiento para determinar si ciertos conjuntos de fórmulas son satisfechas.

La idea es poner la *KB* y ¬α en CNF (forma normal conjuntiva) y entonces verificar si el conjunto resultante *S* es insatisfecho en la búsqueda para la derivación de la cláusula vacía.

El procedimiento no distingue entre cláusulas que vienen de la *KB* y aquella que viene de la negación de α, la cual llamamos consulta (query).

Observe que si tenemos un número de consultas que queremos realizar para la misma *KB*, solamente necesitamos convertir la *KB* a CNF y añadir cláusulas negadas para cada consulta.

Adicionalmente, si queremos añadir un nuevo hecho α a la KB se puede hacer añadiendo las cláusulas para cada α en la *KB*. Por lo tanto, para usar este tipo de procedimiento de consecuencia lógica es bueno cuidar la *KB* en FNC, añadiendo y removiendo cláusulas como sea necesario.

16.3.4 Algoritmo de Unificación

En lógica proposicional, es fácil determinar que dos literales no pueden ser verdad a la misma vez. Por ejemplo para *L* y ¬*L*. En lógica de predicados, este proceso de igualación es más complicado, ya que la unión de variables debe ser considerado. Por ejemplo, *hombre(Enrique)* y ¬*hombre(Enrique)* es una contradicción, mientras que *hombre(Enrique)* y ¬*hombre(Oscar)* no lo es. Por lo tanto, para determinar contradicciones, se requiere de un proceso de igualación que compare literales y descubra donde existe un conjunto de substituciones que hace ellas idénticas. Este procedimiento es llamado *Algoritmo de Unificación*.

Las reglas de igualación son simples. Si dos constantes, o dos funciones, o dos predicados son idénticos, la igualdad es cierta, de otra manera es falsa. Una variable, puede igualar a otra variable, cualquier constante o una función o un predicado, con la restricción de que la función o predicado no puede contener cualquier instancia de la variable siendo igualada.

La única complicación de este procedimiento es que se puede encontrar una simple y consistente substitución para la literal entera, no separada para cada pieza de ella. Por ejemplo, suponga que se quieren unificar las expresiones:

(P x x)

(P w z)

Las dos instancias de *P* igualan. Enseguida compara *x* y *w*, y decide que si se substituye *w* por *x*, ellas igualan. Se escribe esta substitución como: *w*/*x*.

Se continúa y se igualan *x* y *z*, produciéndose la substitución *z*/*x*. Pero no se puede sustituir *w* y *z* por *x*, de modo que no se produce una substitución consistente. De tal manera que después de la primera substitución *y*/*x*, se hace esa substitución en las piezas remanentes de las literales, dando;

(y)

(z)

Ahora se intenta unificar estas literales, lo cual tiene éxito con la substitución *z/y*.

16.3.5 Resolución en Lógica de Predicados

Con la unificación es fácil determinar si dos literales son contradictorias, enseguida se muestra el algoritmo de resolución para lógica de predicados, asumiendo un conjunto de estatutos dados *F* y un estatuto a ser probado *S*:

1. Seleccione dos cláusulas. Llame a éstas las cláusulas padres.
2. Resuelva ellas juntas. La resolvente será la disyunción de todas las literales de ambas cláusulas con la apropiada substitución realizada y con la siguiente excepción: Si hay un par de literales *T1* y ¬*T2* tal que una de las cláusulas padres contiene *T1* y la otra contiene *T2* y si *T1* y *T2* son unificables, ninguna de las dos aparecerán en la resolvente. Llamaremos T1 y T2 literales complementarias. Use la substitución producida por la unificación para crear la resolvente.
3. Si la resolvente es la cláusula vacía, entonces una contradicción ha sido encontrada. Si no, entonces añada al conjunto de cláusulas disponibles para el procedimiento.

Si la elección de las cláusulas a resolver es hecha de manera sistemática, entonces el procedimiento de resolución encontrará una contradicción si existe. Sin embargo, puede tomar un tiempo grande. Existen estrategias para hacer que la elección pueda acortar el tiempo del proceso, por ejemplo :

- Resuelva pares de cláusulas que contengan literales complementarias.
- Eliminar cláusulas tautológicas.
-

16.4 Correlación con el o los temas y subtemas del programa de estudio vigente

Tema: Fundamentos de programación lógica.

Subtemas: Unificación y resolución, subcláusulas de Horn y resolución SLD.

16.5 Material y equipo necesario

Equipo de cómputo e Internet.

16.6 Metodología

Las actividades de aprendizaje que se consideran en esta práctica son las siguientes:

Usando el principio de resolución y la correspondiente *KB*, probar la conclusión que se establece en cada uno de los siguientes ejercicios:

1. Considere los siguientes axiomas:

   a) Todos los coyotes cazan algún correcamino.

   b) Cada correcamino que dice “beep-beep” es inteligente.

c) Ningún coyote atrapa algún correcamino.

d) Cualquier coyote que caza algún correcamino pero no lo atrapa esta frustado.

Conclusión: Si todos los correcaminos dicen “beep-beep”, entonces todos los coyotes están frustados.

2. Considere los siguientes axiomas:

   a) Cualquiera que maneje una Harley es de rudo carácter.

   b) Cada motociclista maneja algo que es una Harley o una BMW.

   c) Cualquiera que maneje cualquier BMW es un yuppie.

   d) Cada yuppie es un abogado.

   e) Cualquier niña bonita no sale con alguien que sea de rudo carácter.

   f) María es una niña bonita y Juan es un motociclista.

   Conclusión: Si Juan no es un abogado entonces María no sale con Juan.

3. Considere los siguientes axiomas:

   a) Cada inversionista compra algo que esta en la bolsa o en bonos.

   b) Si el Promedio Down-Jones cae, entonces todas las bolsas que no están en oro también caen.

   c) Si la tasa de interés T-Bill sube, entonces todos los bonos caen.

   d) Cada inversionista que compró algo que cae no es feliz.

   Conclusión: Si el Promedio Down-Jones cae y la tasa de interés T-Bill sube, entonces cualquier inversionista que esta feliz compró alguna bolsa en oro.

4. Considere los siguientes axiomas:

   a) Todos los niños aman a todos los dulces.

   b) Cualquiera que ama cualquier dulce no es un fanático de la nutrición.

   c) Cualquiera que come cualquier calabaza es un fanático de la nutrición.

   d) Cualquiera que compra cualquier calabaza la corta o la come.

   e) Juan compra una calabaza.

   f) La panocha es un dulce.

   Conclusión: Si Juan es un niño, entonces Juan corta alguna calabaza.

16.7 Sugerencias didácticas

Realizar ejercicios de la traducción de la lógica de predicados al formato clausular.

16.8 Evidencias de aprendizaje (Reporte del alumno)

Reporte de la práctica de acuerdo al Anexo A.

16.9 Bibliografía

C.L. Chang & R.C. Lee, (1973), Symbolic Logic and Mechanical Theorem Proving, Academic Press.

Programación Lógica

Apuntes y Prácticas

Práctica 17: Fundamentos de Programación Lógica

17.1 Competencias a desarrollar

Identificar los elementos de la programación lógica.

Comprender el proceso de deducción de la programación lógica.

Resolución de problemas.

Capacidad de análisis y síntesis.

Habilidad para buscar y analizar información proveniente de fuentes diversas.

Habilidades de investigación.

Capacidad de aprender.

17.2 Objetivo de la Práctica

Conocer los elementos de un programa lógico.

17.3 Introducción

El cambio cualitativo producido a inicios de los años sesenta en el concepto de la programación introduciendo la metodología de diseño estructurado y la verificación de programas fue un primer paso de replanteamiento del diseño de los procesos : pasa de expresar el <*como*> resolver un problema, a expresar el <*qué*> (conocimiento para resolver un problema). En efecto, un programa en el que se intercalan afirmaciones para verificación incorpora ambos aspectos:

- Los predicados intercalados y finales reflejan la forma de entender el proceso como conjunto de afirmaciones sobre los estados sucesivos de información generados por el proceso.
- Los bloques de programación entre afirmaciones definen la forma de computar, consistente con el conjunto de afirmaciones.

Los desarrollos en el área de la deducción automática han dado lugar al inicio de una línea en que se plantea la simple especificación lógica de los procesos prescindiendo de la especificación de procedimientos, quedando la parte procedural embebida en un software de interpretación de las especificaciones de la misma forma

en que los compiladores de los lenguajes tradicionales suponen para el usuario la traducción al lenguaje de máquina.

El origen de estos métodos son las técnicas de extracción de respuestas de los procesos de resolución iniciados por Green en 1969 y seguidos por Colmerauer en 1972 y por Kowalski en 1974, han dado lugar a la programación lógica, una línea del desarrollo de software, válida tanto para la especificación de algoritmos como para modelos de inteligencia artificial.

En 1975 Roussel y Colmerauer presentaron un sistema de obtención automática de respuestas que, desde un punto de vista lógico es de ámbito restringido, pero que completado con algunas operaciones características constituye un verdadero vehículo de programación en Lógica y como tal se denominó el mencionado sistema: lenguaje Prolog (Programación en Lógica). Kowalski hizo notar que el sistema puede considerarse como un demostrador automático por resolución lineal restringido a un tipo especial de cláusulas, propuestas por Horn en 1951.

Una cláusula de Horn es una fórmula clausular en donde aparece, a lo más, una literal afirmada, como se muestra a continuación:

$\neg A(x) \vee \neg D(x) \vee \neg E(x) \vee F(x,y)$

$\neg A(x) \vee \neg R(x,y)$

$\neg A(x), F(x,y)$

Por analogía con la formulación implícita y deductiva la notación utilizada para este tipo de cláusulas es una lista de literales negadas como antecedente y la literal afirmado como consecuente separadas por el signo de implicación. Así los ejemplos anteriores se formulan :

$A(x), D(x), E(x) \rightarrow F(x,y)$

$A(x), R(x,y) \rightarrow$

$A(x) \rightarrow$

$\rightarrow F(x,y)$

Con esta notación la regla de resolución tiene el aspecto de la regla de corte empleada en los sistemas de deducción natural. Así la cláusula resolvente de :

$A(x),D(y) \rightarrow R(x,y)$ y $R(x,z),P(x) \rightarrow S(z)$ es $A(x),D(z),P(z) \rightarrow S(z)$

Se demuestra que todo problema resoluble en cláusulas genéricas tiene un modelo resoluble en cláusulas Horn por lo que las restricciones de notación no significan una pérdida de capacidad de representación, en cambio facilitan la formulación de técnicas de resolución.

Procesos de Obtención de Respuestas en Cláusulas de Horn

Si se tiene un conjunto de cláusulas de Horn representativo del conocimiento sobre un tema, la forma de obtener una respuesta es intentar obtener una instancia de la pregunta deducida de las cláusulas iniciales en la que aparezcan las variables especificadas.

Si se hace abstracción del proceso de unificación, la forma de resolver un problema es la siguiente:

- Se introduce el conjunto de cláusulas-premisa ordenadas *i* en la forma:

Pi, Qi, Ri, Si → Ti

- Se introduce el conjunto de objetivos a demostrar en la forma:

O1,O2,O3,....,On

El proceso de resolución consiste en:

a) Seleccionar un objetivo en el orden en que se presentan. Por tanto, en el caso del ejemplo se seleccionaría *O1*.
b) Se busca en cada etapa, con objetivo *Oi* la primera de las premisas en que aparece una instancia del predicado *Oi* unificable con él.

   Si aparece:

   Pk,Qk,Rk,Sk → Tk

   Y *Tk* se unifica en *Oi*, se incluye en la lista de objetivos la premisas *Pk, Qk, Rk, Sk*, una vez aplicada la substitución unificadora, que ahora son los primeros de la lista.

c) Si se llega a una situación en que un objetivo no es unificable con alguna de las conclusiones de cada cláusula, el proceso termina sin solución en esa rama y vuelve a optar por otra solución de unificación de las existentes en b) para el último objetivo ensayado.
d) Si se llega a dejar vacía la lista de objetivos, el proceso termina. Una solución se define por los valores de las variables libres existentes en los objetivos.
e) En la etapa b), puede haber diversas opciones de unificación de un operador, el sistema, salvo que se especifique algo en contra, vuelve a reconsiderar las opciones pendientes de unificación para generar nuevas ramas, de forma que el proceso lista todas las respuestas posibles.

El proceso antes descrito junto con:

- Un sistema manejador de archivos de cláusulas.
- Un conjunto de cláusulas implícito que incorpora conocimiento general sobre aritmética y relaciones típicas como =, etc.
- Un conjunto de símbolos o funciones:
  - Para cortar o dirigir el proceso de búsqueda.

- Para dirigir el proceso de unificación.

Constituye un sistema *Prolog*.

Otra forma de justificar el procedimiento Prolog no basada en técnicas deductivas, es desde un punto de vista de técnica de resolución de problemas, es un tipo de sistema de reglas de producción que realmente representan procedimientos de descomposición.

En efecto, dada una cláusula Horn: A,B,C,D → E

Puede entenderse de una forma deductiva: Si ocurre A,B,C,D debe ocurrir E.

Y también puede entenderse de forma procedural: para resolver el problema E es preciso resolver antes los problemas A,B,C, y D.

Entendido de esta segunda forma, la orientación de la flecha de la fórmula sería contraria:

E → A,B,C,D

y escrito de esta forma el conjunto de premisas representa los procedimientos básicos de descomposición de los posibles objetivos de un problema en otros, definiéndose las condiciones que se asumen como ciertas, como aquellas que no requieren descomposición:

G →

que es un correlato del concepto de fórmula válida en lógica:

→ G

Los sistemas Prolog optan por este tipo de formulación presentada por Kowalski, por estar más cerca del concepto clásico de programación y, por tanto, ser más accesible a las personas no habituadas a la lógica.

Al presentar esta propuesta Kowalski hacía notar que la definición de un algoritmo se lleva a efecto mediante dos componentes expresados convencionalmente por la ecuación:

$$ALGORITMO = LOGICA + CONTROL$$

El conjunto de premisas representa la lógica del algoritmo que, como se ha tratado, representa las posibles formas de descomponer una serie de objetivos en otros. La componente de control constituye la estrategia de selección de opciones de descomposición y de regreso.

17.4 Correlación con el o los temas y subtemas del programa de estudio vigente

Tema: Fundamentos de programación lógica.

Subtema: Programación lógica con clausulas de Horn y semántica de los programas lógicos.

17.5 Material y equipo necesario

Equipo de cómputo e Internet.

17.6 Metodología

Las actividades de aprendizaje que se consideran en esta práctica son las siguientes:

1. Investigar acerca de la máquina abstracta de Warren.
2. Descargar e instalar el interprete SWI-Prolog de la siguiente dirección:

   http://www.swi-prolog.org/download/stable

17.7 Sugerencias didácticas

Realizar la mayor cantidad de ejercicios de traducción de la lógica de predicados al formato clausular de Horn.

17.8 Evidencias de aprendizaje (Reporte del alumno)

Reporte de la práctica de acuerdo al Anexo A.

Interprete SWI-Prolog instalado.

17.9 Bibliografía

C.L. Chang & R.C. Lee, (1973), Symbolic Logic and Mechanical Theorem Proving, Academic Press.

R. Brachman & H. Levesque, (2004), Knowledge Representation and Reasoning, Elsevier Science & Technology Books.

J. F. Sowa, (1999), Knowledge Representation: Logical, Philosophical, and Computational Foundations, Brooks Cole Publishing Co.

Programación Lógica

Apuntes y Prácticas

Práctica 18: Introducción al lenguaje Prolog

18.1 Competencias a desarrollar

Identificar los elementos del lenguaje Prolog.

Resolución de problemas.

Capacidad de análisis y síntesis.

Habilidad para buscar y analizar información proveniente de fuentes diversas.

Habilidades de investigación.

Capacidad de aprender.

18.2 Objetivo de la Práctica

Solucionar problemas usando el lenguaje Prolog.

18.3 Introducción

Un programa en Prolog está formado por un conjunto de proposiciones (cláusulas de Horn) en donde cada una ellas puede ser de las formas siguientes:

a) Hecho o aserción
b) Regla o implicación
c) Pregunta

Los hechos y las reglas responden a la forma general siguiente:

$$B(T1,T2,....,Tn) :- A1,\ A2,....,Ap$$

Donde $p >= 0$ y *:-* es el símbolo para la implicación.

Cada *Ai* al igual que *B* es una relación seguida de sus argumentos (una lista de términos). Un término es una constante, una variable o una función seguida de sus argumentos.

De lo anterior notamos que la programación en Prolog consiste en la declaración de reglas acerca de objetos y sus relaciones, y preguntas acerca de objetos y sus relaciones, esto es, consiste de una base de datos que contiene los hechos y reglas que son relevantes al problema.

El problema es entonces resuelto formulando una pregunta a ser contestada por el sistema usando la información contenida en la base de datos, por esta razón, Prolog es un sistema conversacional.

18.3.1 Hechos

Estos declaran cosas o sucesos que son siempre verdad, son cláusulas que tienen únicamente la parte izquierda o cabeza de la cláusula, o simplemente es una oración sencilla, su forma es la siguiente :

$$B(T1, T2, \ldots, Tn).$$

Donde se puede ver que la notación usada es la estándar "Edimburgo", la cual probablemente es la más común. En esta notación, los nombres de todas las relaciones y objetos inician con minúsculas, la relación es escrita primero y los objetos son escritos separados por comas y encerrados entre paréntesis. Un punto finaliza la cláusula.

Es necesario que los objetos aparezcan en orden consistente, el orden particular no es tan importante como la consistencia, por ejemplo, en lenguaje natural los siguientes hechos serían escritos de esta manera:

Pedro es el padre de Juan.

Sócrates es humano.

El oro es un metal.

En Prolog estos hechos serían escritos así:

padre(pedro,juan).

humano(sócrates).

metal(oro).

Los nombres padre, humano y metal son relaciones, llamados también predicados y los objetos o argumentos son pedro, juan, sócrates y oro.

18.3.2 Preguntas

Una vez que se tiene una base de datos que viene a ser la interpretación declarativa, se puede activar un procedimiento cuando se produce una pregunta como la siguiente:

?-metal(oro).

Efectuándose una computación/inferencia para construir/deducir una respuesta, en este caso afirmativa, considerando el anterior conjunto de hechos.

Las preguntas se resuelven por un encadenamiento hacia atrás de las reglas (en términos de procedimientos, una sustitución de la llamada al procedimiento por el cuerpo del procedimiento).

Las preguntas pueden consistir de uno o más objetivos a lograr. Se habla de objetivos porque tratarán de ser satisfechos al igualarse con cláusulas de la base de conocimiento con el mismo predicado y los mismos argumentos. Cuando se tienen más de un objetivo significa la conjunción o disyunción de objetivos; por ejemplo:

?- metal(oro) y humano(sócrates).

Donde se intentará satisfacer ambos objetivos para obtener la respuesta. En este caso, considerando los hechos, la respuesta resulta afirmativa, ya que se encuentran cláusulas que igualan a las cláusulas de la pregunta.

18.3.3 Reglas

Se usan para indicar que un hecho depende de otro grupo de hechos o reglas. Una regla es un estatuto general acerca de objetos y sus relaciones y consiste de un cuerpo y una cabeza. El cuerpo es el lado derecho de la cláusula y es la condición, la cabeza es el lado izquierdo de la cláusula y es la conclusión.

B(T1,T2,....,Tn) :- A1, A2,....,Ap

La conclusión y la condición son conectados por un símbolo que indica implicación, por ejemplo:

X es un pájaro si X es un animal y X tiene plumas.

Con las reglas también se expresan definiciones como en el ejemplo anterior, lo cual en Prolog podría ser escrito así:

pájaro(X) :- animal(X), plumas(X).

Donde notamos que la cabeza de la regla describe qué hecho intenta definir y el cuerpo de la regla describe la conjunción de subobjetivos que deberán ser satisfechos, uno después de otro, para que la cabeza de la regla sea cierta.

En la anterior regla se introduce el concepto de variable, indicado por X, lo cual se define a continuación.

Una variable en Prolog puede representar cualquier objeto y se distingue de los demás objetos ya que inicia con mayúscula. Cuando una variable aparece en una pregunta puede significar que, quién, todos. Pero es importante considerar que las variables pueden aparecer en cualquier tipo de cláusula.

18.3.4 Búsqueda de Soluciones

De forma general, la evaluación de una pregunta:

?- A(T1,T2,...,Tn).

Se hace intentando aplicar cada una de las cláusulas que tengan el mismo nombre o predicado, y la lista de argumentos deben ser idénticas, si no lo son, un método de unificación, encuentra automáticamente la sustitución adecuada, si es que existe.

La derivación por encadenamiento de reglas o en términos computacionales, la evaluación mediante ejecución de llamadas sucesivas a procedimientos requiere una cierta estrategia, ya que se presentan dos casos en donde hay que elegir entre diferentes alternativas :

Cuando hay varias cláusulas con las que podemos identificar nuestra pregunta, cuál de ellas seleccionar.

Si la pregunta se compone de varias relaciones, cuál de ellas resolver primero.

Prolog tiene una estrategia fija, las cláusulas se aplican en el orden en que aparecen, y para las preguntas siempre es de izquierda a derecha.

Cuando sucede un fracaso (no se puede seguir aplicando cláusulas) se efectúa un regreso para explorar la siguiente posibilidad, y se obtienen todas las soluciones.

Considere los siguientes hechos y reglas.

grande(oso).

grande(elefante).

pequeño(gato).

café(oso).

negro(gato).

gris(elefante).

oscuro(Z) :- negro(Z).

oscuro(Z) :- café(Z).

Pregunta: ?- oscuro(X), grande(X).

?- oscuro(X), grande(X).

Ejecución:

La lista inicial de objetivos es: *oscuro(X), grande(X).*

Busca en el programa desde el inicio de una cláusula cuya cabeza iguale al primer objetivo oscuro(X). La cláusula 7 es encontrada: oscuro(Z) si negro(Z). Reemplaza el primer objetivo por el cuerpo instanciado de la cláusula 7, dando una nueva lista de objetivos.

Busca en el programa para encontrar una cláusula con negro(X). La cláusula 5; negro(gato) satisface el objetivo y se instancia X=gato.

Busca en el programa la nueva lista de objetivos: grande(gato). No se encuentra cláusula, sin embargo regresa al paso 3 y se deshace la instancia X=gato. Ahora la lista de objetivos es otra vez: negro(X), grande(X). Continua buscando en la base de conocimientos abajo de la cláusula 5. No se encuentra cláusula, se regresa al paso 2 y continua abajo de la cláusula 7. La cláusula 8 es encontrada : oscuro(Z) :- café(Z).

Reemplaza el primer objetivo en la lista de objetivos por café(X), dando: café(X) , grande(X).

Busca en la base de datos café(X) encontrando café(oso). Esta cláusula no tiene cuerpo, de modo que la lista de objetivos se limita ahora a: grande(oso), donde X es instanciada a oso.

Busca en el programa y encuentra la cláusula grande(oso) de tal manera que la lista de objetivos es satisfecha. Esto indica terminación con éxito y la correspondiente instanciación de variables es: X=oso.

Lo cual sería la respuesta del sistema a la pregunta hecha.

18.4 Correlación con el o los temas y subtemas del programa de estudio vigente

Tema: Fundamentos de programación lógica.

Subtema: Representación clausular del conocimiento. Consulta de una base de conocimiento.

18.5 Material y equipo necesario

Equipo de cómputo e Internet.

18.6 Metodología

Las actividades de aprendizaje que se consideran en esta práctica es la solución de los siguientes ejercicios:

1. Asuma que se tiene definida la siguiente base de hechos de relaciones familiares:

   progenitor(clara,jose).

   progenitor(tomas, jose).

   progenitor(tomas,isabel).

   progenitor(jose, ana).

   progenitor(jose, patricia).

   progenitor(patricia,jaime).

   a. Realice un grafo dirigido para representar estas relaciones.
   b. Construya un programa en Prolog para realizar las siguientes preguntas:

      ?-progenitor(jaime,X).

      ?-progenitor(X,jaime).

      ?-progenitor(clara,X), progenitor(X,patricia).

      ?-progenitor(tomas,X), progenitor(X,Y), progenitor(Y,Z).

   c. Establezca en lenguaje natural el significado de las preguntas.

d. Construya para cada pregunta el espacio de búsqueda.

2. Construya en Prolog las siguientes preguntas y obtenga los resultados con respecto a la base de hechos mencionada anteriormente.
   a. ¿Quién es el progenitor de Patricia?
   b. ¿Tiene Isabel un hijo o una hija?
   c. ¿Quién es el abuelo de Isabel?
   d. ¿Cuáles son los tíos de Patricia? (no excluir al padre).

3. Dada la base de hechos familiar del ejercicio 1, y suponiendo definidas las siguientes cláusulas:

   hombre(X).

   mujer(X).

   progenitor(X,Y).

   dif(X,Y):- X \=Y.

   Donde las 3 primeras cláusulas se definirán como hechos, sustituyendo las variables *X* e *Y* por constantes según corresponda (por tanto no se podrá poner una variable como argumento, ya que una variable haría que el hecho fuera cierto para cualquier objeto) y la última como una regla (donde el símbolo \= significa distinto). Escribir las reglas de Prolog que expresen las siguientes relaciones:

   a. es_madre(X).
   b. es_padre(X).
   c. es_hijo(X).
   d. hermana_de(X,Y).
   e. abuelo_de(X,Y) y abuela_de(X,Y).
   f. hermanos(X,Y). Tener en cuenta que una persona no es hermano de sí mismo.
   g. tia(X,Y). Excluir a los padres.

18.7 Sugerencias didácticas

Revisar la sintaxis y semántica del lenguaje Prolog.

18.8 Evidencias de aprendizaje (Reporte del alumno)

Reporte de la práctica de acuerdo al Anexo A.

18.9 Bibliografía

L. Sterling, E. Shapiro, (1994), The Art of Prolog, Second Edition: Advanced Programming Techniques, The MIT Press.

P. Julian, M. Alpuente, (2007), Programación Lógica. Teoría y Práctica. Pearson Prentice Hall.

M. A. Covington, D. Nute, A. Vellino, (1996), Prolog Programming in Depth, 1st Edition, Prentice Hall.

Práctica 19: Recursividad en Prolog

19.1 Competencias a desarrollar

Identificar los elementos de la programación lógica y aplicar la programación lógica.

Resolución de problemas.

Capacidad de análisis y síntesis.

Habilidad para buscar y analizar información proveniente de fuentes diversas.

Habilidades de investigación.

Capacidad de aprender.

19.2 Objetivo de la Práctica

Solucionar problemas usando recursividad en Prolog.

19.3 Introducción

Cuando escribimos un método para resolver un problema en particular, una de las técnicas básicas de diseño es dividir la tarea en subtareas más pequeñas. Por ejemplo, el problema de agregar (o multiplicar) *n* enteros consecutivos se puede reducir a un problema de agregar (o multiplicar) n-1 enteros consecutivos:

```
1 + 2 + 3 +... + n = n + [1 + 2 + 3 + .. + (n-1)]

1 * 2 * 3 *... * n = n * [1 * 2 * 3 * .. * (n-1)]
```

Por lo tanto, si introducimos un método *sumaR(n)* (o *vecesR(n)*) que agrega (o multiplica) enteros de *1* a *n*, entonces las expresiones aritméticas anteriores pueden reescribirse como:

```
sumaR(n) = n + sumaR(n-1)

vecesR(n) = n * vecesR(n-1)
```

Dicha definición funcional se denomina definición recursiva, ya que la definición contiene una llamada a sí misma. En cada llamada recursiva, el argumento de *sumaR(n)* (o *vecesR(n)*) se reduce en uno. Lleva n-1 llamadas hasta que lleguemos al

caso base; esto es parte de una definición que no hace una llamada a sí mismo. Cada definición recursiva requiere casos base para evitar la recursión infinita.

La programación recursiva está directamente relacionada con la inducción matemática El caso base es demostrar que la afirmación es cierta para algunos valores o valores específicos de *N*. El paso de inducción: suponga que una declaración es verdadera para todos los enteros positivos menores que *N*, luego demuestre que es verdadera para *N*.

En matemáticas y también en la computación nos enfrentamos a la solución de problemas que por su naturaleza tienen una definición recursiva, por ejemplo; el calculo del factorial de un numero entero o la definición de un árbol binario, entre otros.

Particularmente en la programación declarativa el mecanismo de la recursividad es muy utilizado ya que no se tiene la estructura iterativa, de tal manera que el problema se resuelve a través de la recursividad.

Un algoritmo es llamado recursivo si resuelve un problema reduciéndolo a una instancia del mismo problema con una entrada pequeña. Un ejemplo clásico de la recursividad es el cálculo del factorial de un número entero. Esta función esta definida como:

$$n! = \begin{cases} 1 \; si \; n=1 \\ (n*(n-1)! \;\; si \; n>1 \end{cases}$$

La función del cálculo del factorial puede ser escrita como:

entero Factorial(n)

if (n==1) entonces

regresa 1

de otra manera

*regresa (n*Factorial(n-1))*

La recurrencia del factorial del número 3 es la siguiente:

3! = 3 * (3-1)!

= 3 * 2!

= 3 * 2 * (2-1)!

= 3 * 2 * 1!

= 3 * 2 * 1 * (1-1)!

= 3 * 2 * 1 * 0!

= 3 * 2 * 1 * 1

= 6

Si la llamada recursiva se produce al final de un método, se denomina recursividad final. La recursividad de cola es similar a un lazo. El método ejecuta todas las instrucciones antes de pasar a la siguiente llamada recursiva. Si la llamada recursiva se produce al inicio de un método, se llama recursividad principal. El método guarda el estado antes de saltar a la siguiente llamada recursiva.

Una de las grandes potencialidades del Prolog está en la definición de reglas recursivas. Por ejemplo, cuando se requiere la definición de relaciones generales que impliquen el uso de jerarquías, el siguiente caso ilustra este concepto.

Asuma que se tiene la relación predecesor entre dos personas:

predecesor (X, Y)

como se puede apreciar en la figura 11 hay relaciones directas y relaciones transitivas.

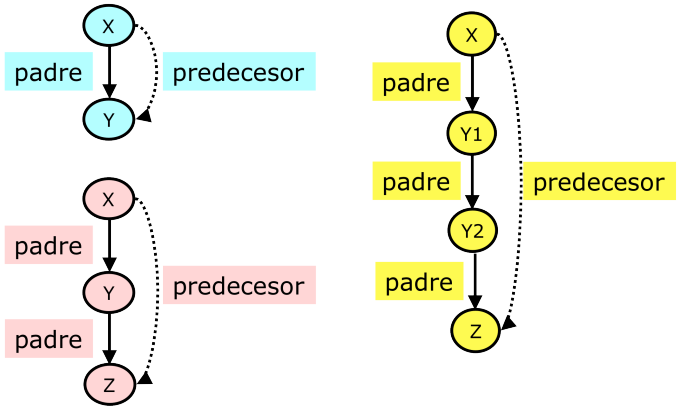


Figura 11. La relación predecesor

De tal manera que se puede definir como una generalización como se muestra a continuación en la figura 12.

Para toda X y Z,

X es un predecesor de Z si

Hay una Y tal que

(1) X es un padre de Y y

(2) Y es un predecesor de Z.

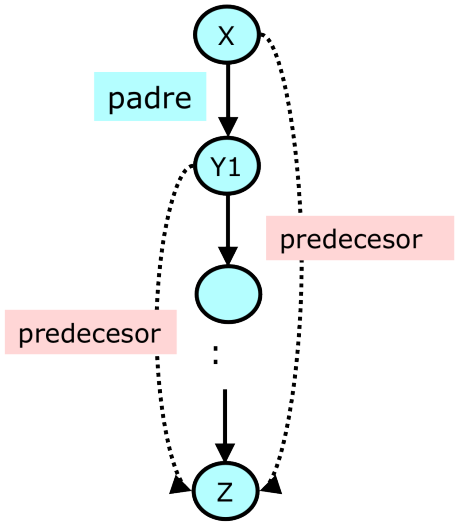


Figura 12. La relación predecesor definida como una generalización

En la definición se puede apreciar que es recursiva, por lo tanto se puede escribir como una regla recursiva en Prolog:

predecesor(X, Z):- padre(X, Z).

predecesor(X, Z):- padre(X, Y), predecesor(Y, Z).

19.4 Correlación con el o los temas y subtemas del programa de estudio vigente

Tema: Fundamentos de programación lógica.

Subtema: Representación clausular del conocimiento. Consulta de una base de conocimiento.

19.5 Material y equipo necesario

Equipo de cómputo e Internet.

19.6 Metodología

Las actividades de aprendizaje que se consideran en esta práctica es la solución de los siguientes ejercicios:

1. Escriba un programa en Prolog para obtener el factorial de un numero entero.
2. Fibonacci nació en el año 1170 en Pisa, Italia, y murió en 1250. Su verdadero nombre fue Leonardo Pisano. En 1202, escribió un libro: Liber Abbaci, que significa "Libro de cálculo".

   El número de Fibonacci se define como la suma de los dos números precedentes:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

Escriba un programa en Prolog para obtener la secuencia Fibonacci de longitud *n*.

Escriba un programa en Prolog que calcule el resto de una división de numero enteros. Por ejemplo, una llamada *resdiv(100,12)* regresará el valor 4 (debido a que 100 dividido por 12 es igual a 8 con resto 4).

4. Asuma que se tiene definida la siguiente base de hechos de relaciones familiares:

   progenitor(clara,jose).

   progenitor(tomas, jose).

   progenitor(tomas,isabel).

   progenitor(jose, ana).

   progenitor(jose, patricia).

   progenitor(patricia,jaime).

   Escribir las reglas de Prolog que expresen las siguientes relaciones:

   a. *abuelo_de(X,Y).*
   b. *bisabuelo_de(X,Y).*

19.7 Sugerencias didácticas

Realizar procesos recursivos de manera manual con los ejemplos presentados en clase.

19.8 Evidencias de aprendizaje (Reporte del alumno)

Reporte de la práctica de acuerdo al Anexo A.

19.9 Bibliografía

L. Sterling, E. Shapiro, (1994), The Art of Prolog, Second Edition: Advanced Programming Techniques, The MIT Press.

P. Julian, M. Alpuente, (2007), Programación Lógica. Teoría y Práctica. Pearson Prentice Hall.

M. A. Covington, D. Nute, A. Vellino, (1996), Prolog Programming in Depth, 1st Edition, Prentice Hall.

Práctica 20: Estructuras y Listas en Prolog

20.1 Competencias a desarrollar

Identificar los elementos de la programación lógica y aplicar la programación lógica.

Resolución de problemas.

Capacidad de análisis y síntesis.

Habilidad para buscar y analizar información proveniente de fuentes diversas.

Habilidades de investigación.

Capacidad de aprender.

20.2 Objetivo de la Práctica

Conocer y aplicar las estructuras y listas en la programación declarativa.

20.3 Introducción

Los objetos estructurados (o estructuras) en Prolog son términos de la forma *f(t1, ..., tn)* donde *f* es un símbolo de función n-ario, o funtor, y *t1, ..., tn* son a su vez términos (que pueden ser constantes, variables o a su vez estructuras). Ejemplos de estructuras son las siguientes:

libro(prolog, autor(ivan, bratko))

*a+(b*c)* ó *+(a, *(b, c))*

Las estructuras se suelen representar por árboles donde el funtor es un nodo y los componentes son los subárboles que cuelgan de dicho nodo. Las expresiones anteriores se representan de la siguiente forma como se muestra en la figura 13.

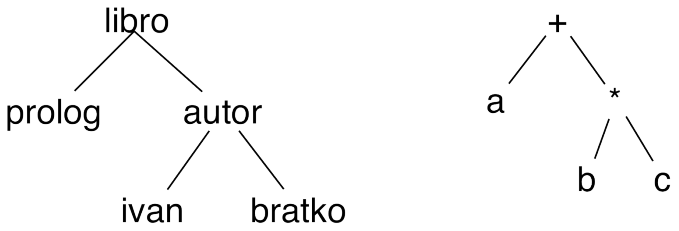
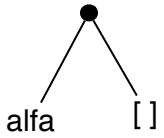


Figura 13. Representación de estructuras

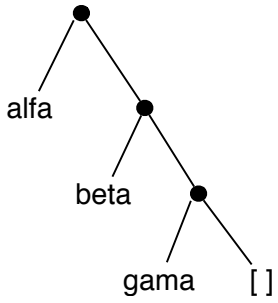
Una lista es una secuencia ordenada de elementos que puede tener cualquier longitud. Los elementos de una lista (como los de cualquier otra estructura) son términos, los cuales pueden ser en particular otras listas.

Las listas pueden representarse como un tipo especial de árbol. Una lista es o bien la lista vacía, sin elementos, denotada [ ], o bien una estructura cuyo funtor se denota "." (punto) y con dos componentes llamados "cabeza" y "resto" de la lista.

Una lista con un solo elemento es *.(alfa, [ ])* y su árbol es:



Una lista con un varios elementos es *.(alfa,.(beta,.(gama,[ ])))* y su árbol es:



Pero la forma sintáctica más habitual de escribir las listas es separando los elementos entre comas y toda la lista encerrada entre corchetes. Las listas anteriores se escriben de la siguiente forma: *[alfa]* y *[alfa,beta,gama]*.

La forma de manejar las listas es dividiéndolas en cabeza (el primer argumento del funtor punto o el primer elemento de la lista) y resto (el segundo argumento del funtor punto o la lista formada por el resto de elementos de la lista).

La forma sintáctica en Prolog para describir una lista con cabeza *X* y resto *Y* es la notación *[X|Y]*.

20.4 Correlación con el o los temas y subtemas del programa de estudio vigente

Tema: Fundamentos de programación lógica.

Subtema: Programación lógica con números, listas y árboles.

20.5 Material y equipo necesario

Equipo de cómputo e Internet.

20.6 Metodología

Las actividades de aprendizaje que se consideran en esta práctica es la solución de los siguientes ejercicios:

1. Usando estructuras y listas en Prolog realice lo siguiente:

    a. Representar la información relativa a las siguientes familias:

    En la primera familia,

    • el padre es Juan Pérez Pérez, nacido el 7 de Mayo de 1960, trabaja de profesor y gana 600 pesos diarios;

    • la madre es Sofía López López, nacida el 10 de marzo de 1962, trabaja de médica y gana 900 pesos diarios;

    • el hijo es Juan Pérez López, nacido el 5 de Enero de 1980, estudiante;

    • la hija es María Pérez López, nacida el 12 de Abril de 1992, estudiante.

    En la segunda familia,

    • el padre es José Ruiz Ruiz, nacido el 6 de Marzo de 1963, trabaja de pintor y gana 1200 pesos diarios;

    • la madre es Luisa Gálvez Gomez, nacida el 12 de Mayo de 1964, trabaja de médica y gana 900 pesos diarios;

    • un hijo es José Luis Ruiz Gálvez, nacido el 5 de Febrero de 1990, estudiante;

    • una hija es María José Ruiz Gálvez, nacida el 12 de Junio de 1992, estudiante;

    • otro hijo es José María Ruiz Gálvez, nacido el 12 de Julio de 1994, estudiante.

    b. Realizar las siguientes consultas:

    ¿existe familia sin hijos?

    ¿existe familia con un hijo?

    ¿existe familia con dos hijos?

    ¿existe familia con tres hijos?

¿existe familia con cuatro hijos.?

c. Buscar los nombres de los padres de familia con tres hijos.
d. Definir la relación

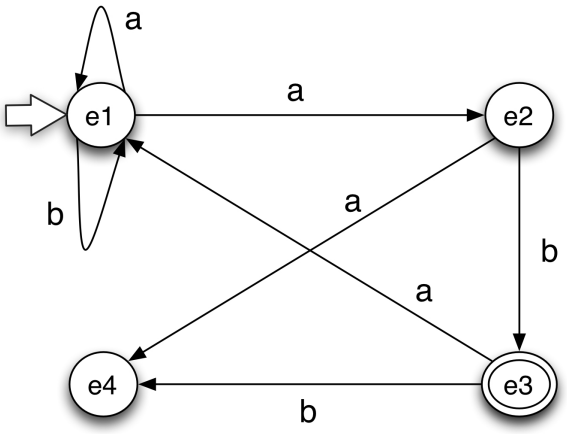
casado(X) que se verifique si X es un hombre casado.

e. Preguntar por los hombres casados.
f. Definir la relación

casada(X) que se verifique si X es una mujer casada.

g. Preguntar por las mujeres casadas.
h. Determinar el nombre de todas las mujeres casadas que trabajan.
i. Definir la relación hijo(X) que se verifique si X figura en alguna lista de hijos.
j. Preguntar por los hijos.
k. Definir la relación persona(X) que se verifique si X es una persona existente en la base de conocimiento.
l. Preguntar por los nombres y apellidos de todas las personas existentes en la base de datos.
m. Determinar todos los estudiantes nacidos antes de 1993.
n. Definir la relación fecha_de_nacimiento(X,Y) de forma que si X es una persona, entonces Y es su fecha de nacimiento.
o. Buscar todos los hijos nacidos en 1992.
p. Definir la relación sueldo(X,Y) que se verifique si el sueldo de la persona X es Y.
q. Buscar todas las personas nacidas antes de 1964 cuyo sueldo sea superior a 720 pesos diarios.
r. Definir la relación total(L,Y) de forma que si L es una lista de personas, entonces Y es la suma de los sueldos de las personas de la lista L.
s. Calcular los ingresos totales de cada familia.

2. Consideremos el autómata representado por el siguiente grafo.



Siendo e1 estado inicial y e3 el estado final.

a. Representar el autómata utilizando las siguientes relaciones:

   final(X) que se verifica si X es el estado final.

   trans(Q1,X,Q2) que se verifica si se puede pasar del estado Q1 al estado Q2 usando la letra X.

   nulo(Q1,Q2) que se verifica si se puede pasar del estado Q1 al estado Q2 mediante un movimiento nulo.

b. Definir la relación acepta(E,L) que se verifique si el autómata, a partir del estado E, acepta la lista L.

   Por ejemplo, ?- acepta(e1,[a,a,a,b]).

c. Determinar los estados a partir de los cuales el autómata acepta la lista [a,b].
d. Determinar las palabras de longitud 3 aceptadas por el autómata a partir del estado e1.
e. Definir la relación acepta_acotada_1(E,L,N) que se verifique si el autómata, a partir del estado E, acepta la lista L y la longitud de L es N.
f. Buscar las cadenas aceptadas a partir de e1 con longitud 3.
g. Definir la relación acepta_acotada_2(E,L,N) que se verifique si el autómata, a partir del estado E, acepta la lista L y la longitud de L es menor o igual que N.
h. Buscar las cadenas aceptadas a partir de e1 con longitud menor o igual 3.

20.7 Sugerencias didácticas

Discutir con los alumnos si el lenguaje Prolog permite listas heterogéneas, esto es, con datos de diferentes tipos y cuál sería su aplicación si lo permite el lenguaje.

20.8 Evidencias de aprendizaje (Reporte del alumno)

Reporte de la práctica de acuerdo al Anexo A.

20.9 Bibliografía

L. Sterling, E. Shapiro, (1994), The Art of Prolog, Second Edition: Advanced Programming Techniques, The MIT Press.

P. Julian, M. Alpuente, (2007), Programación Lógica. Teoría y Práctica. Pearson Prentice Hall.

M. A. Covington, D. Nute, A. Vellino, (1996), Prolog Programming in Depth, 1st Edition, Prentice Hall.

Práctica 21: Corte en Prolog

21.1 Competencias a desarrollar

Identificar los elementos de la programación lógica y aplicar la programación lógica.

Resolución de problemas.

Capacidad de análisis y síntesis.

Habilidad para buscar y analizar información proveniente de fuentes diversas.

Habilidades de investigación.

Capacidad de aprender.

21.2 Objetivo de la Práctica

Conocer y aplicar las técnicas de control de búsqueda en Prolog.

21.3 Introducción

Prolog es capaz de dar todas las posibles respuestas a una pregunta, es decir, buscará todas las posibles refutaciones SLD (estrategias de resolución). El espacio de búsqueda de soluciones se puede representar mediante un árbol, donde cada rama representa una posible refutación SLD. La exploración del árbol de soluciones PROLOG, es en profundidad, es decir, explora una rama del árbol hasta llegar al final de ella (a un éxito o fracaso) y sólo después recorrerá las siguientes ramas de la misma forma. Para comprender mejor esto, considere el siguiente programa en Prolog.

```
padre(juan,pedro).
padre(pedro,maria).

abuelo(X,Y) :- padre(X,Z),padre(Z,Y).
```

Si se realiza la siguiente pregunta en el interprete de Prolog se obtiene la traza de la ejecución de la pregunta que se muestra posteriormente.

```
?- trace,abuelo(X,Y),notrace.

The debugger will first creep -- showing everything (trace)
   1    1  Call: abuelo(_636,_637) ?
```

```
    2    2  Call: padre(_636,_717) ?
    2    2  Call: padre(_636,_717) ?
    2    2  Call: padre(_636,_717) ?
    2    2  Exit: padre(juan,pedro) ?
    3    2  Call: padre(pedro,_637) ?
    3    2  Exit: padre(pedro,maria) ?
    1    1  Exit: abuelo(juan,maria) ?
The debugger is switched off
```

Si dibujamos el árbol de búsqueda de manera que las distintas posibilidades de resolución de cada objetivo se dibujen ordenadamente de izquierda a derecha, tal y como se muestra en la figura 14, entonces el orden de exploración de las ramas es de izquierda a derecha.

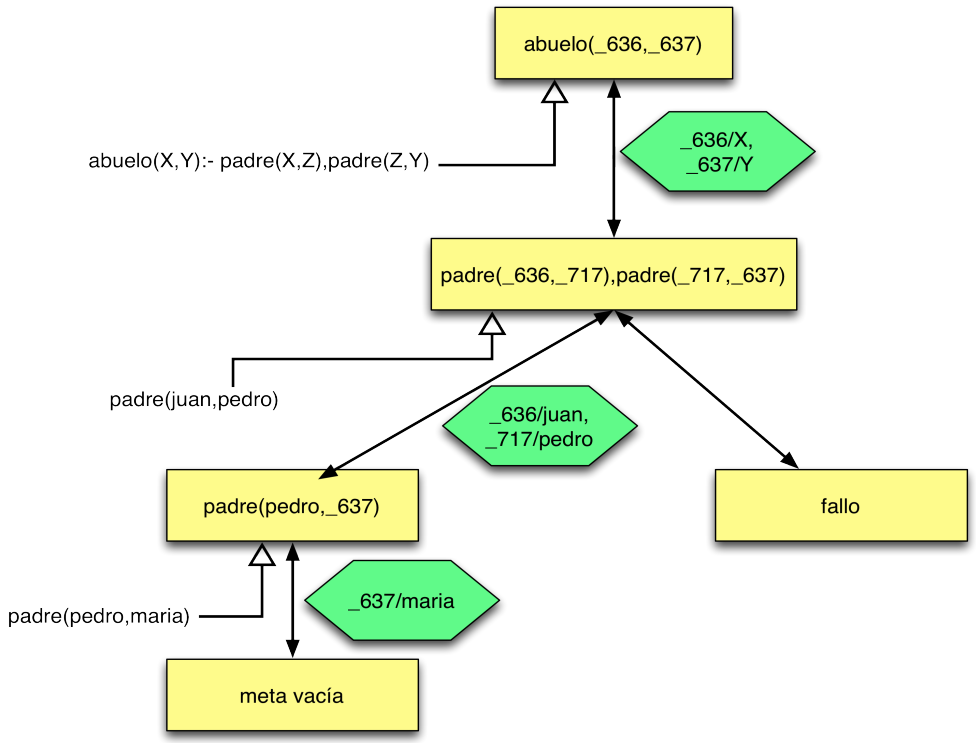


Figura 14. Árbol de búsqueda

El árbol de búsqueda de soluciones para una pregunta y un programa, representa todos los caminos que el sistema debe recorrer para encontrar las respuestas a la pregunta.

Pero, ¿Qué ocurre cuando se llega al final de la rama?

Si ha sido un éxito (se obtiene) corresponde a una respuesta del sistema.

Si ha sido un fracaso se debe seguir buscando por otra rama hasta encontrar una refutación. El proceso a seguir es hacer una vuelta atrás por los nodos recorridos hasta llegar a alguno del que salga otra rama. Entonces se recorre esta rama en profundidad y así sucesivamente.

Por tanto, el recorrido en el árbol es en ***profundidad*** y con ***vuelta atrás*** (backtracking) a la última elección hecha. Este proceso de vuelta atrás para recorrer más ramas del árbol, no sólo se hace para encontrar la primera refutación (que corresponde a la primera respuesta del sistema), sino también si se quieren encontrar otras posibles refutaciones (otras posibles respuestas).

Para controlar el proceso de búsqueda, esto es, de alguna manera podar el árbol de búsqueda, el Prolog tiene un mecanismo para realizarlo, esto es el predicado predefinido llamado el ***corte*** (cut). Este predicado previene la vuelta atrás.

El lenguaje Prolog hace backtracking automáticamente, si es necesario para satisfacer un objetivo. El backtracking automático es un concepto de programación útil, porque alivia al programador de la carga de la programación con backtracking explícita. Por otro lado, el backtracking incontrolado puede causar ineficiencia en un programa. Además, algunas veces es preferible controlar el backtracking para no hacerlo. Podemos hacer esto utilizando el corte.

El corte como menciono previamente es un predicado predefinido, cuya sintaxis es *"!"* y cuyo comportamiento es el siguiente. Suponemos definido el predicado *"alfa"* con las siguientes cláusulas:

alfa :- beta, !, gama.

alfa :- delta.

Para comprobar o demostrar que el predicado *alfa* es cierto, pasamos a comprobar si es cierto el predicado *beta*. Pueden ocurrir dos cosas:

1. Si *beta* no es cierto, mediante backtracking, se comprobará si *delta* es cierto. Si *delta* es cierto, quedará demostrado que *alfa* es cierto. Hasta aquí el predicado *alfa* tiene un comportamiento normal en Prolog.

2. Si *beta* es cierto, se sobrepasa la "barrera" del corte que significa:

   - Que no se podrá hacer backtracking para tomar otras posibilidades alternativas en el predicado *beta*.

   - Que no se pueden considerar otras posibles definiciones del predicado a (es decir, no podemos acceder a *alfa :- delta*.).

   Si *gama* es cierto, entonces se ha demostrado *alfa*, pero si *gama* no se cumple, entonces el predicado a falla.

Adicionalmente, el corte también extiende el poder de expresividad de Prolog y permite la definición de un tipo de negación: la negación como fallo y la asocia con "la asunción de un mundo cerrado".

En la asunción de mundo cerrado (CWA) se asume que todos los hechos y reglas válidos ya han sido incluidos como parte del programa, por lo que cualquier cosa que no se deduce del mismo es falsa.

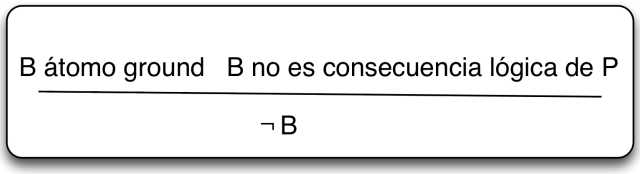


Figura 15. Regla de CWA

En la figura 15 se puede apreciar la regla de CWA, el problema es cómo determinar que B no es consecuencia lógica de un programa (puede haber árboles SLD con ramas infinitas).

La negación por fallo es una versión restringida de la regla de CWA. Se dice que un árbol SLD falla finitamente si es finito y no tiene ramas de éxito.

Dado un programa P el conjunto de falla finita de P es

{B | B átomo ground y existe un árbol SLD que falla finitamente con : ¬B como raíz}

Lo anterior es menos poderoso que CWA pero es efectivo y además es computable. Por ejemplo, considere el siguiente programa:

estudiante(jose).

estudiante(beto).

estudiante(jaime).

profesor(maria).

La negación como fallo nos permite deducir que **maria** no es un estudiante.

En Prolog, el fallo se implementa con el predicado *fail* que siempre produce fallo. Es útil cuando queremos detectar casos explícitos que invalidan un predicado. Para evitar la aplicación de una regla, se puede forzar el fallo con una combinación del corte, y la constante "*fail*", es un objetivo que nunca se satisface.

Por ejemplo: "Todos los pájaros, excepto el avestruz y el pinguïno, vuelan":

vuela(X) :- pinguino(X), !, fail.

vuela(X) :- avestruz(X), !, fail.

vuela(X) :- pajaro(X).

En cuanto a la negación, existe un predicado de negación en Prolog (*not*) que está implementado como negación por fallo. Esto quiere decir que se evalúa como falso

cualquier cosa que Prolog sea incapaz de verificar que su predicado argumento es cierto.

Ejemplo:

no_nulo(X) :- not (X =:= 0).

Ejemplo:

saldo_cuenta(flora,3000000).

saldo_cuenta(antonio,2000000).

millonario(X) :- saldo_cuenta(X, Y), Y > 1000000.

pobre(X) :- not millonario(X).

Consultas:

?- millonario(X).

X = flora ;

X = antonio

?- pobre(X).

No.

La negación en Prolog puede definirse como:

```
not(P) :-
        P, !, fail
        ;
        true.
```

Muchos intérpretes de Prolog incluyen el predicado *not* predefinido, con un operador asociado \+.

Ejemplo:

vuela(X) :- pajaro(X), \+ pinguino(X), \+ avestruz(X).

Sin embargo, se tienen los siguiente problemas con el corte y la negación; en el corte los programas ya no corresponden a la definición declarativa: el orden de las cláusulas importa, y puede ser necesario forzar a que algún argumento sea una variable no instanciada.

En la negación no corresponde a una negación lógica, sino al hecho de que no hay evidencia para demostrar lo contrario.

En Prolog, una consulta con una variable no instanciada se satisface si hay al menos una asignación de valores a la variable que la cumpla:

buena_comida(X) -> X = casa_paco

Al usar la negación, esa consulta pasa a ser cierta si el argumento de la negación fue falso, es decir, si ninguna asignación posible de valores cumplió la fórmula:

not(razonable(X)) = no(existe X tal que X razonable) = para todo X, X no es razonable.

Ejemplo:

Cada ciudadano de Sonora es un ciudadano de México.

Cada ciudadano de Arizona no es un ciudadano de México.

Cada ciudadano de Sonora no es un ciudadano de USA.

Cada ciudadano de Arizona es un ciudadano de USA.

La gobernadora Pavlovich es un ciudadano de Sonora.

El gobernador Ducey es un ciudadano de Arizona.

Pruebe que la gobernadora Pavlovich es un ciudadano de México y no lo es de USA, así como el gobernador Ducey es un ciudadano de USA pero no lo es de México.

En Prolog:

```
ciudadano(X,mexico) :- ciudadano(X,sonora).
ciudadano(X,usa) :- ciudadano(X,arizona).
no_ciudadano(X,mexico) :- ciudadano(X,arizona).
no_ciudadano(X,usa) :- ciudadano(X,sonora).
ciudadano(pavlovich,sonora).
ciudadano(ducey,arizona).
```

21.4 Correlación con el o los temas y subtemas del programa de estudio vigente

Tema: Fundamentos de programación lógica.

Subtema: Control de búsqueda en programas lógicos.

21.5 Material y equipo necesario

Equipo de cómputo e Internet.

21.6 Metodología

Las actividades de aprendizaje que se consideran en esta práctica es la solución de los siguientes ejercicios:

1. Considere el siguiente programa en Prolog:

   padre(juan, antonio).

padre(juan, luis).

padre(luis, pedro).

abuelo(X, Y) : padre(X, Z), padre(Z, Y).

a. Construya el árbol de búsqueda para la siguiente pregunta:

   abuelo(juan , X).

b. Realice la siguiente pregunta y analice el resultado.

   ?- trace, abuelo(juan , X), notrace.

2. Considere las siguientes reglas:

   R1: si X<3 entonces Y=0

   R2: si 3<=X y X<6 entonces Y=2

   R3: si 6>=X entonces Y=4

   a. Construya un programa en Prolog para las reglas anteriores, de tal manera que se pueda hacer la siguiente pregunta:

      ?- f(1,Y), 2<Y.

      ¿Cuál es el resultado obtenido?

   b. Modifique su programa considerando el uso del corte para hacer mas eficiente el programa. Realice la misma pregunta y analice sus resultados.

3. Considere la siguiente base de hechos:

   animal(coco).
   animal(pepita).
   serpiente(pepita).

   a. Construya un programa en Prolog que incluya la base de hechos y la siguiente regla usando *cut-fail* y *not*:
      "A Paty le gustan los animales, excepto las serpientes".
   b. Analice sus resultados para las preguntas para cada caso implementado de la regla:
      ?- gusta(paty, coco).
      ?- gusta(paty, pepita).
      ?- gusta(paty, X).

21.7 Sugerencias didácticas

Construir espacios de búsqueda para pequeños ejemplos de programas lógicos.

21.8 Evidencias de aprendizaje (Reporte del alumno)

Reporte de la práctica de acuerdo al Anexo A.

21.9 Bibliografía

L. Sterling, E. Shapiro, (1994), The Art of Prolog, Second Edition: Advanced Programming Techniques, The MIT Press.

P. Julian, M. Alpuente, (2007), Programación Lógica. Teoría y Práctica. Pearson Prentice Hall.

M. A. Covington, D. Nute, A. Vellino, (1996), Prolog Programming in Depth, 1st Edition, Prentice Hall.

Práctica 22: Predicados predefinidos (built-in) en Prolog

22.1 Competencias a desarrollar

Identificar los elementos de la programación lógica y aplicar la programación lógica.

Resolución de problemas.

Capacidad de análisis y síntesis.

Habilidad para buscar y analizar información proveniente de fuentes diversas.

Habilidades de investigación.

Capacidad de aprender.

22.1 Objetivo de la Práctica

Conocer y aplicar los predicados predefinidos (built-in) en Prolog.

22.2 Introducción

Los predicados predefinidos son aquellos que su definición es proporcionada por el sistema Prolog, en lugar de sus propias clasusulas. Estos predicados pueden proporcionar funcionalidades que no son obtenidas por las definiciones de un Prolog puro. Por ejemplo, funcionalidades como la entrada y salida.

Estos predicados predefinidos pueden tener efectos colaterales. Esto es, satisfacer un objetivo puede causar cambios además de la instanciación de los argumentos. Esto, por supuesto, no puede ocurrir con un predicado definido en el Prolog puro.

Otro importante hecho acerca de los predicados predefinidos es que ellos pueden esperar diferentes tipos de argumentos. Por ejemplo, considere el predicado <, definido de modo que X<Y tiene éxito si el numero X es menor que el numero Y. Tal relación no puede ser definido en Prolog sin alguna ayuda externa que sabe algo acerca de los números.

De modo que < es proporcionado como un predicado predefinido y su definición involucra el uso de alguna subyacente operación de máquina para probar la el tamaño comparativo de los números. ¿Que podría pasar si se introduce un objetivo “menor que” donde X es un átomo, o si ambos X e Y no están instanciados?. La definición en términos de la máquina simplemente no aplica. De modo que se debe

estipular que X<Y es solo un objetivo sensible si ambos X e Y están instanciados a números cuando un intento es hecho para satisfacerlo. Cuando esta condición no esta implementada en el sistema Prolog, una posibilidad es que el objetivo simplemente falla, otra posibilidad es que un mensaje de error es impreso y el sistema toma alguna acción apropiada como se muestra en la figura 16.

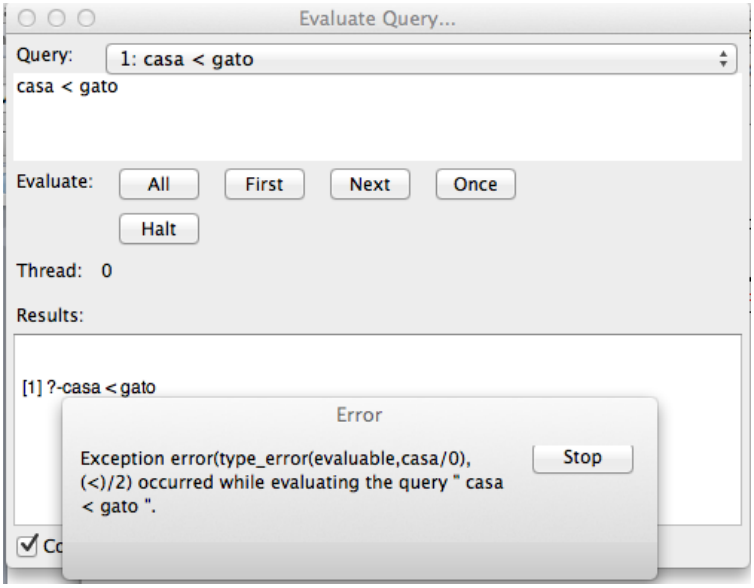


Figura 16. Mensaje de error al usar el predicado definido <.

Como se mencionó previamente hay predicados predefinidos para la entrada y salida, así como para el manejo de listas, para la clasificación de términos, para agregar o remover clausulas de manera dinámica al programa, para el manejo de términos, para afectar el retroceso (backtracking), para crear objetivos complejos, para evaluar expresiones aritméticas, para comparar números, para depurar el programa, entre otros.

En la siguiente liga se pueden consultar los predicados predefinidos de SWI-Prolog:

http://www.swi-prolog.org/pldoc/man?section=builtin

22.3 Correlación con el o los temas y subtemas del programa de estudio vigente

Tema: Fundamentos de programación lógica.

Subtema: Manipulación de términos. Predicados metalógicos.

22.4 Material y equipo necesario

Equipo de cómputo e Internet.

22.5 Metodología

Las actividades de aprendizaje que se consideran en esta práctica es la solución de los siguientes ejercicios:

1. Escriba un programa en Prolog para verificar si una lista conteniendo enteros es un palíndromo.
2. Considere la siguiente base de hechos:

   nombre(0,cero).
   nombre(1,uno).
   nombre(2,dos).
   nombre(3,tres).
   nombre(4,cuatro).
   nombre(5,cinco).
   nombre(6,seis).
   nombre(7,siete).
   nombre(8,ocho).
   nombre(9,nueve).

   Escriba un programa en Prolog para la relación traduccion(L1,L2) que verifique si L2 es la lista de palabras correspondientes a los dígitos de la lista L1. Por ejemplo:

   ?- traduccion([2,8],L).

   L = [dos,ocho]

   a. Considere una primer solución por *recursión en L1*.
   b. Considere una segunda solución usando el predicado predefinido *findall*.
   c. Considere una tercera solución usando el predicado predefinido *maplist*.
   d.
3. Escriba un programa en Prolog que escriba en pantalla la lista L de códigos ASCII en forma de cadena de caracteres, por ejemplo:

   ?- escribe_cadena([80,114,111,108,111,103]).
   Prolog
   yes

4. Definir una base de datos que relaciones comidas con sus ingredientes. Además se definirá otra relación que describe los ingredientes que se disponen y otras dos relaciones: *puedo_cocinar(Comida)* que se satisfará para una comida si se dispone de todos los ingredientes y *necesita_ingrediente(Comida,Ingrediente)* que se satisfará para una comida y un ingrediente si la comida contiene el ingrediente.
   Añadir a la base de datos la cantidad disponible de cada ingrediente y también la cantidad de cada ingrediente necesaria para cada comida y modificar la relación *puedo_cocinar(Comida)* para que compruebe que se puede cocinar esa comida si se dispone de la cantidad necesaria de cada ingrediente.

22.6 Sugerencias didácticas

Discutir con los alumnos la diferencia entre predicados lógicos y predicados metalógicos.

22.7 Evidencias de aprendizaje (Reporte del alumno)

Reporte de la práctica de acuerdo al Anexo A.

22.8 Bibliografía

L. Sterling, E. Shapiro, (1994), The Art of Prolog, Second Edition: Advanced Programming Techniques, The MIT Press.

P. Julian, M. Alpuente, (2007), Programación Lógica. Teoría y Práctica. Pearson Prentice Hall.

M. A. Covington, D. Nute, A. Vellino, (1996), Prolog Programming in Depth, 1st Edition, Prentice Hall.

Bibliografía

Sestoft Peter, (2017), Programming Language Concepts, Springer International Publishing.

Kurt Will, (2018), Get Programming with Haskell , Manning Publications.

Allen Christopher, Moronuki Julie, (2016), Haskell Programming: From First Principles, Gumroad.

Thomasson Samuli, (2016), Haskell High Perfomance Programming, Packt Publishing Limited.

Church James, Learning Haskell Data Analysis, Packt Publishing Limited.

Bird Richard, (2014), Thinking Functionally with Haskell, Cambridge University Press.

Rose-Gómez César Enrique, (2012), Administración y Organización de Datos, Tecnológico Nacional de México / Instituto Tecnológico de Hermosillo.

Irnazo P.J., (2005), Lógica Simbólica para Informáticos, Alfaomega – Rama.

Chang C.L., Lee R.C., (1973), Symbolic Logic and Mechanical Theorem Proving, Academic Press, 1973.

Brachman R., Levesque, H., (2004), Knowledge Representation and Reasoning, Elsevier Science & Technology Books.

Suples E., (1988), Lógica Matemática, Reverté.

Sowa J.F., (1999), Knowledge Representation: Logical, Philosophical, and Computational Foundations, Brooks Cole Publishing Co.

Sterling L., Shapiro, (1994), The Art of Prolog, Second Edition: Advanced Programming Techniques, The MIT Press.

Huth M., Ryan M., (2004), Logic in Computer Science: Modelling and Reasoning about Systems, Cambridge University Press; 2a edition.

Newborn M., (2000), Automated Theorem Proving: Theory and Practice, Springer; 1st. Edition.

ANEXO A

Elaboración de reportes de prácticas

El contenido de un reporte de práctica debe contener los siguientes apartados:

1. **Portada** que debe incluir los datos que se muestran el documento del anexo B. El formato de la hoja será el vigente en el instituto.
2. **Introducción** (apoyada en una revisión de literatura).
3. **Objetivo** (u objetivos).
4. **Materiales y métodos** realizados en laboratorio.
5. **Resultados y Discusión** (resultados de la práctica que se discuten con la revisión de literatura).
6. **Conclusiones** que deben ser puntuales, obtenidas de los resultados discutidos con literatura.
7. **Referencias bibliográficas** de acuerdo al formato IEEE, ver anexo C.

Los cuadros y figuras incluidos deben citarse en el texto con numeración progresiva, además de contener su respectiva leyenda, en el caso de los cuadros en la parte superior y las figuras en la parte inferior.

Los materiales y métodos son aquellos utilizados en la práctica, no lo que viene en la descripción de la práctica, además debe estar redactada en tiempo pretérito.

La discusión se hace con base en los resultados obtenidos y la revisión de literatura. Las conclusiones es la única sección que puede ir puntualizada.

En el caso de que la práctica incluya el desarrollo de programas de cómputo, se debe incluir en el reporte las pantallas de la ejecución del programa y se debe enviar el código y el reporte de la práctica al correo electrónico y/o plataforma de acuerdo a lo establecido por el profesor.

ANEXO B

SEP
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO

Instituto Tecnológico de Hermosillo

NOMBRE DE LA MATERIA

PRÁCTICA No. N

NOMBRE DE LA PRÁCTICA

NOMBRE DEL ALUMNO
CARRERA
NUMERO DE CONTROL

NOMBRE DEL MAESTRO

FECHA

ANEXO C

Formato IEEE para las citas y referencias bibliográficas

Las referencias se numeran una a una como van a pareciendo en el texto. No se puede usar un número para un grupo de referencias. No hay sección de Bibliografía que no haya sido citada.

IEEE no exige mencionar al autor y título en el texto. Se inserta en la gramática de la oración como si fueran notas al pie o sustantivos.

Ejemplo notas:
como demuestra Brown [4], [5]; y fue mencionado antes [2], [4]–[7], [9];
Smith [4] y Brown y Jones [5]; Wood et al. [7]

Tres o más autores se pone el primero y "et al." (abreviación de et alii, "y otros" en latín).
Ejemplo sustantivos: como se demuestra en [3]; y de acuerdo a [4] y [6]–[9].

Publicaciones periódicas:
Formato básico
[1] J. K. Author, "Name of paper," Abbrev. Title of Periodical, vol. x, no. x, pp. xxx-xxx, Abbrev. Month, year.
Ejemplo
[1] R. E. Kalman, "New results in linear filtering and prediction theory," J. Basic Eng., ser. D, vol. 83, pp. 95-108, Mar., 1961.

Libros:
Formato básico:
[1] J. K. Author, "Title of chapter in the book," in Title of His Published Book, xth ed. City of Publisher, Country if not USA: Abbrev. of Publisher, year, ch. x, sec. x, pp. xxx–xxx.-
Ejemplos:
[1] B. Klaus and P. Horn, Robot Vision. Cambridge, MA: MIT Press, 1986.
[2] L. Stein, "Random patterns," in Computers and You, J. S. Brake, Ed. New York: Wiley, 1994, pp. 55-70.

Informes:
Formato básico:
[1] J. K. Author, "Title of report," Abbrev. Name of Company, City of Co., Abbrev. State, Rep. xxx, year.
Ejemplos:
[1] E. E. Reber, R. L. Michell, and C. J. Carter, "Oxygen absorption in the earth's atmosphere," Aerospace Corp., Los Angeles, CA, Tech. Rep. TR-0200 (4230-46)-3, Nov. 1988.
[2] J. H. Davis and J. R. Cogdell, "Calibration program for the 16-foot antenna," Elect. Eng.

Res. Lab., Univ. Texas, Austin, Tech. Memo. NGL-006-69-3, Nov. 15, 1987.

Memorias Publicadas de Conferencias:

Se puede omitir el año si el mismo está incluido en el nombre.

Formato básico:

[1] J. K. Author, "Title of paper," in Unabbreviated Name of Conf., City of Conf., Abbrev. State (if given), year, pp. xxx-xxx.

Ejemplos:

[1] G. R. Faulhaber, "Design of service systems with priority reservation," in Conf. Rec. 1995 IEEE Int. Conf. Communications, pp. 3–8.

[2] S. P. Bingulac, "On the compatibility of adaptive controllers," in Proc. 4th Annu. Allerton Conf. Circuit and System Theory, New York, 1994, pp. 8–16.

Artículos presentados a Conferencias:

Formato básico:

[1] J. K. Author, "Title of paper," presented at the Unabbrev. Name of Conf., City of Conf., Abbrev. State, year.

Ejemplos:

[1] J. G. Kreifeldt, "An analysis of surface-detected EMG as an amplitude-modulated noise," presented at the 1989 Int. Conf. Medicine and Biological Engineering, Chicago, IL.

[2] G. W. Juette and L. E. Zeffanella, "Radio noise currents on short sections on bundle conductors," presented at the IEEE Summer Power Meeting, Dallas, TX, June 22-27, 1990, Paper 90 SM 690-0 PWRS.

Patentes:

Formato básico:

[1] J. K. Author, "Title of patent," U.S. Patent x xxx xxx, Abbrev. Month, day, year.

Ejemplo:

[1] J. P. Wilkinson, "Nonlinear resonant circuit devices," U.S. Patent 3 624 125, July 16, 1990.

Tesis de Maestría y Doctorado:

Formatos básicos:

[1] J. K. Author, "Title of thesis," M.S. thesis, Abbrev. Dept., Abbrev. Univ., City of Univ., Abbrev. State, year.

[2] J. K. Author, "Title of dissertation," Ph.D. dissertation, Abbrev. Dept., Abbrev. Univ., City of Univ., Abbrev. State, year.

Ejemplos:

[1] N. Kawasaki, "Parametric study of thermal and chemical nonequilibrium nozzle flow," M.S. thesis, Dept. Electron. Eng., Osaka Univ., Osaka, Japan, 1993.

[2] J. O. Williams, "Narrow-band analyzer," Ph.D. dissertation, Dept. Elect. Eng., Harvard Univ., Cambridge, MA, 1993.

Apuntes de clases:
Formato básico:
[1] "Título de los apuntes o materia", class notes for Código de la asignatura, Departamento, Institución o Universidad, época y año.
Ejemplo:
[12] "Signal integrity and interconnects for high-speed applications", class notes for ECE497-JS, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Winter 1997.

Estándares:
Formato básico:
[1] Title of Standard, Standard number, date.
Ejemplos:
[1] IEEE Criteria for Class IE Electric Systems, IEEE Standard 308, 1969.
[2] Letter Symbols for Quantities, ANSI Standard Y10.5-1968.

Libros Electrónicos:
Formato básico:
[1] J. K. Author. (year, month day). Title (edition) [Type of medium]. volume(issue). Available: site/path/file
Ejemplos:
[1] S. Khutaina. (1995, Aug. 15). EMBASE handbook (3rd ed.) [Online]. 3(21). Available: Knowledge Index File: EMBASE Handbook (EMHB)
[2] J. Jones. (1991, May 10). Networks (2nd ed.) [Online]. Available: http://www.atm.com

Artículos de publicaciones periódicas digitales:
Formato básico:
[1] J. K. Author. (year, month). Title. Journal [Type of medium]. Volumen (issue), paging if given. Available FTP/WWW: Directory: File:
Ejemplos:
[1] R. P. Drew. (1996, Jan.). All-digital oversampled front-end sensors. Science Online [Online]. 3(1). Available FTP: sci.mit.edu Directory: pub/journals/sci.online/issue12 File: 012bel5.txt
[2] M. Semilof. (1996, July 15). Driving commerce to the Web—Corporate Intranets and the Internet: Lines blur. Communications Week [Online]. 6(19). Available: http://www.techweb.com/se/directlink.cgi?CWK19960715S0005